UNIVERSITÀ DI PISA

Facoltà di Scienze, Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

TESI DI LAUREA

Un editore grafico di alberi logici e fisici per il DBMS relazionale JRS

RELATORE

Prof. Antonio ALBANO

Candidato

Ciro VALISENA

ANNO ACCADEMICO 2009-2010

Sommario

L'algebra relazionale e i piani di accesso per eseguire le interrogazioni sono fra gli argomenti principali trattati nei corsi su basi di dati. I DBMS disponibili, commerciali o per scopi didattici, non consentono di fare interrogazioni usando l'algebra relazionale e mostrano i piani di accesso generati dall'ottimizzatore per interrogazioni scritte in SQL. Si presenta un'estensione del DBMS relazionale JRS con un editore grafico per formulare ed eseguire interrogazioni sia con l'algebra relazionale che con un piano di accesso usando alberi logici o fisici. L'estensione proposta rende il JRS uno strumento particolarmente adatto per l'apprendimento di concetti ritenuti importanti nello studio delle basi di dati.

INDICE

Capitolo 1

INTRODUZIONE

Nell'uso interattivo dei RDBMS le interrogazioni vengono di solito formulate in un linguaggio dichiarativo noto come **SQL**. In alcuni casi è possibile visualizzare l'algoritmo generato dall'ottimizzatore per eseguire l'interrogazione richiesta, che farà uso delle strutture di memorizzazione disponibili e di operatori su di esse. Questo algoritmo è noto come *piano di accesso*. La visualizzazione del piano di accesso e di varie informazioni relative agli operatori, in alcuni casi comprende anche dei suggerimenti o avvertimenti dell'ottimizzatore[1], sono utili per la progettazione fisica della base di dati al fine di migliorarne l'efficienza, e generalmente danno importanti indicazioni per la scelta degli indici. Gli operatori di un piano di accesso corrispondono agli algoritmi scelti per gli accessi ai dati delle relazioni in memoria permanente, per realizzare gli operatori dell'algebra relazionale. La rappresentazione grafica di un piano di accesso o di un'espressione algebrica consiste in un albero binario, poichè ogni operatore ha al più arietà 2 e sarà l'operando di al più un operatore, mentre ogni operatore con arietà 0 è una foglia dell'albero.

Nessun RDBMS noto invece mostra la rappresentazione ad albero dell'espressione algebrica equivalente all'interrogazione **SQL**, oppure consente di scrivere un'interrogazione usando l'algebra relazionale, sebbene questo aspetto

1. Si veda per esempio SQL Server Management Studio

sia ritenuto importante da ogni testo sulle basi di dati. Un esempio di sistema che realizza un editor di espressioni algebriche in algebra relazionale è "Relationa"[2], con scopo didattico. Purtroppo non mostra l'albero relativo all'interrogazione, ed esegue l'espressione direttamente, senza fare uso di un DBMS, e quindi senza collegamenti a **SQL** o al piano di accesso.

1.1 Contenuto della tesi

Il sistema JRS, sviluppato presso il Dipartimento di Informatica dell'Università di Pisa, per i corsi di basi di dati prevede una semplice interfaccia per l'uso di **SQL** e, su richiesta, mostra una rappresentazione grafica dei piani di accesso generati dall'ottimizzatore. La tesi presenta un'estensione del JRS per definire interrogazioni con un'interfaccia grafica sia con piani logici, usando l'algebra relazionale, sia con piani di accesso, usando gli operatori fisici del JRS, e di eseguire i piani o un loro sottoalbero. Lo scopo è sempre didattico, in quanto si ritiene utile permettere una formulazione guidata da un'interfaccia grafica di un espressione in algebra relazionale o di un piano di accesso, ancora prima di conoscere **SQL**.

Il Capitolo 1 presenta l'introduzione alla tesi.

Il Capitolo 2 presenta l'algebra relazionale e gli operatori fisici del JRS.

Il Capitolo 3 presenta l'editore grafico di alberi logici e fisici.

Il Capitolo 4 contiene un caso d'uso dell'editore.

Il Capitolo 5 mostra una visione generale e una descrizione dell'implementazione dell'editore.

Il Capitolo 6 mostra la modificabilità ed estendibilità del sistema.

2. http://galileo.dmi.unict.it/wiki/relational/doku.php

Capitolo 2

INTERROGAZIONI IN SISTEMI DI BASI DI DATI RELAZIONALI

2.1 L'algebra relazionale

L'algebra relazionale, fondata sulla logica del prim'ordine e sull'algebra degli insiemi, si basa sul concetto matematico di relazione n-aria. Uno schema di relazione R è definito come segue [2]:

Definizione 2.1 Uno schema di relazione $R : \{T\}$ (per prevità $R\{T\}$) è una coppia formata da un nome di relazione R e da un tipo di relazione definito come segue:

- interi, reali, booleani e stringhe sono tipi primitivi.
- $T_1, \ldots, T_n$ sono tipi primitivi e $A_1, \ldots, A_n$ sono etichette distinte, dette *attributi*, allora $(A_1 : T_1, \ldots, A_n : T_n)$ è un *tipo ennupla di grado n*. Due tipi ennupla sono uguali se hanno uguale il grado, gli attributi e il tipo degli attributi con lo stesso nome. L'ordine degli attributi non è significativo.
- se T è un tipo ennupla, allora $\{T\}$ è un *tipo insieme di ennuple* o *tipo relazione*. Due tipi relazione sono uguali se hanno lo stesso tipo ennupla.

Uno *schema relazionale* è costituito da un insieme di *schemi di relazione* e da vincoli di integrità. Un'ennupla $t = (A_1 := V_1, \ldots, A_n := V_n)$ di tipo

$T_j = (A_1 : T_1, \ldots, A_n : T_n)$ è un insieme di coppie (A_i, V_i) con V_i di tipo T_i. Un'*istanza* dello schema $R_j\{T_j\}$, o *relazione*, è un insieme finito di ennuple $\{t_1, t_2, \ldots, t_k\}$ con t_i di tipo T_j.

L'algebra relazionale è definita come un'insieme di operatori su relazioni che danno come risultato altre relazioni. Grazie a questa proprietà di chiusura è possibile annidare più operatori per produrre espressioni complesse. Un'insieme minimo e completo di operatori costituisce l'insieme degli *operatori primitivi*, in funzione dei quali vengono poi definiti gli *operatori derivati*. Nei DBMS inoltre si estende l'algebra con operatori che non possono essere derivati, come per esempio l'operazione di raggruppamento e il calcolo delle funzioni di aggregazione. Inoltre vengono aggiunti operatori che non operano su insiemi, come l'ordinamento o la proiezione con duplicati, perché nei DBMS tali operatori sono poi utilizzati. Gli esempi vengono fatti sulla base di dati **SportDB**, definita nel manuale del JRS e il cui schema è rappresentato in Figura 2.1. Tale base di dati è pensata per gestire un *Tennis Club*.

2.1.1 Proiezione π

$\pi_{A_1}, \ldots, A_m(R)$

La proiezione restituisce una relazione che ha come attributi un sottoinsieme degli attributi di R, con gli stessi valori e tipi di R. Poichè opera su insiemi, la relazione risultante sarà priva di duplicati. È possibile estendere l'operatore per permettere di rinominare gli attributi, e in questo caso il risultato non avrà lo stesso tipo. Esempio: proiettare la relazione **Players** sull'attributo **Name**, cioè i nomi di tutti i giocatori: $\pi_{Name}(Players)$.
La Figura 2.2 mostra l'interrogazione rappresentata ad albero. Si può estendere l'operatore per permettere di rinominare gli attributi, e in questo caso la relazione risultante avrà i nomi degli attributi rinominati.

La proiezione generalizzata estende la proiezione con la possibilità di usare costanti o espressioni aritmetiche nella lista degli attributi. Per comodità si può anche assegnare un'etichetta ad un'espressione con l'operatore AS:

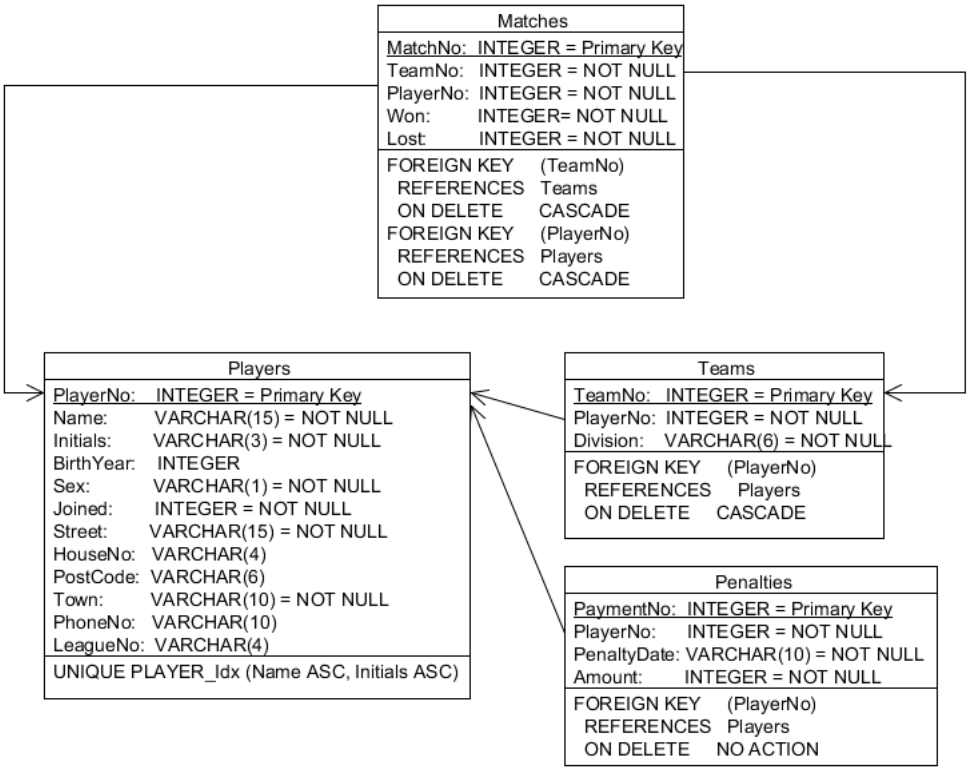

Figura 2.1 Schema della base di dati di esempio

Figura 2.2 Albero per la proiezione

$\pi_{e_1 \text{ AS ide1}, e_2 \text{ AS ide2}, \ldots, e_n \text{ AS ideN}}(E)$,

con $e_1, \ldots, e_n$ espressioni aritmetiche, ottenute a partire da costanti e attributi di E, ed ide1, ..., ideN etichette distinte.

Restrizione σ

$\sigma_\Phi(R) = \{t \mid t \in R \wedge \Phi(t)\}$

Restituisce una relazione dello stesso tipo di R, ma con solo gli elementi di R che soddisfano la condizione Φ. La condizione è una formula proposizionale, e può contenere un confronto tra attributi di R, un confronto tra un attributo di R e una costante o più formule composte con gli operatori logici $\wedge, \vee, \neg$. Se per esempio volessimo i nomi rinominati in N di tutti i giocatori nati dopo il 1950: $\pi_{Name\ AS\ N}(\sigma_{Birthyear>1950}(Players))$.

Figura 2.3 Albero per la restrizione

2.1.2 Operatori insiemistici $\cap, \cup, -$

Gli operatori insiemistici hanno due operandi dello stesso tipo, e restuiscono una relazione del medesimo tipo. Perchè due Relazioni abbiano lo stesso tipo, devono avere gli stessi attributi, con gli stessi tipi degli attributi e nello stesso ordine. Un caso particolare è l'operatore di divisione che non richiede che gli operandi abbiano lo stesso tipo, ma è definito come segue:

$R \div S = \{w \mid \forall s \in S. w \circ s \in R\}$

con XY gli attributi di R e Y gli attributi di S. Il risultato è una relazione con attributi X. Il risultato è l'insieme dei valori degli attributi X in R che sono in relazione con tutti gli attributi Y in S. Per esempio, se R è una relazione

contenente gli esami come candidato, codice (si veda la base di dati StudentiDB) e S una relazione contenente alcuni codici di esami, $R \div S$ restituisce una relazione con gli studenti che hanno superato tutti gli esami elencati in S. L'operatore è derivabile, infatti:

$R \div S \equiv \pi_X(R) - R_1$ con $R_1 = \pi_X((\pi_X(R) \times S) - R)$.

2.1.3 Prodotto $\times$

$R \times S = \{t \circ u \mid t \in R \wedge u \in S\}$

Restituisce una relazione con elementi ottenuti concatenando ogni ennupla di R con quelle di S. Se abbiamo $R(A_1 : T_1, \ldots, A_n : T_n)$ e $S(A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})$ il tipo del risultato è $(A_1 : T_1, \ldots, A_n : T_n, A_{n+1} : T_{n+1}, \ldots, A_{n+m} : T_{n+m})$. Deve il suo nome al prodotto cartesiano ma non restituisce un insieme di coppie di ennuple, ma un insieme di ennuple concantenate. La cardinalità del risultato è data dal prodotto delle cardinalità degli operandi. Gli attributi di R ed S devono essere distinti, e in caso di attributi uguali si deve procedere a una rinominazione prima di poter fare il prodotto.

Esempio: Si vuole calcolare il prodotto tra **Players** e **Matches**. Per prima cosa si rinomina **PlayerNo** della relazione **Matches** in **PlayerInMatch** (ottenendo quindi una nuova relazione)perchè è uguale all'attributo **PlayerNo** di **Players**. Quindi si procede al prodotto.

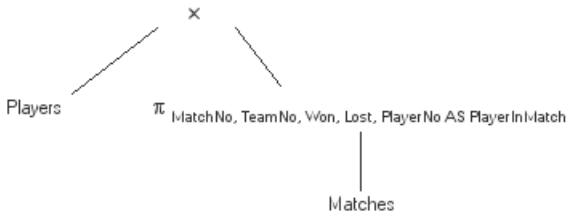

Figura 2.4 Albero per il prodotto

2.1.4 Giunzione $\bowtie$

La giunzione è un operatore derivato, ma viene utilizzata molto spesso nelle interrogazioni. Nella letteratura si distinguono diversi tipi di giunzione, tra cui

quella naturale, *equi join*, θ-join, la semi-giunzione, la giunzione esterna o la giunzione esterna destra o sinistra. Per semplicità ci si limita a considerare la *giunzione naturale*, *l'equi join* e il *θ-join*.

Equi-join e θ-join

Una giunzione del tipo *equi join* è definita come segue:

$$R \underset{A_i=A_j}{\bowtie} S = \{t \circ u \mid t \in R, u \in S, t.A_i = u.A_j\}$$

Si può derivare dagli operatori prodotto e restrizione in questo modo:

$$R \underset{A_i=A_j}{\bowtie} S \equiv \sigma_{A_i=A_j}(R \times S)$$

Restituisce una relazione con elementi la copia delle ennuple del prodotto di R ed S con valori uguali per gli A_i e A_j. La definizione di *θ-join* è analoga però la condizione di restrizione non è limitata all'uguaglianza. Riprendendo il precedente esempio del prodotto, è possibile avere per ogni partita i giocatori che vi hanno giocato:

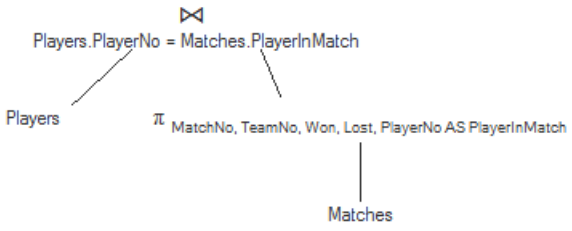

Figura 2.5 Albero con la giunzione al posto del prodotto del precedente esempio

Si noti che i nomi degli attributi sono prefissati dal nome della relazione. Questo evita ambiguità nel caso due attributi abbiano lo stesso nome ma appartengano a due relazioni distinte. Nel caso in cui invece le relazioni non sono distinte, come per esempio nella giunzione di R con se stesso, bisogna procedere a rinominare gli attributi o le relazioni, e in quest'ultimo caso si parla di *variabili di correlazione*. In questo caso si può fare a meno di rinominare, ottenendo un albero più semplice, come visualizzato nella Figura 2.5.

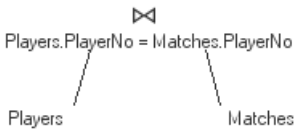

Figura 2.6 Albero per l'equijoin

Giunzione naturale

Nel caso dell'ultima figura, nella condizione di giunzione vi sono due attributi uguali, che verranno riportati due volte nelle ennuple del risultato, come attributi distinti prefissati dal nome della relazione di origine. In casi come questo è possibile effettuare una giunzione naturale, che consiste in un *equi join* con la condizione definita su attributi dallo stesso nome, riportati una sola volta nel risultato. Più precisamente:

$R \bowtie S$

con gli attributi di uguali nomi in R ed S definiti sugli stessi domini (e quindi dello stesso tipo). Sia R(Y, X) ed S(Z, X) con X gli attributi in comune. $R \bowtie S$ restituisce una relazione con attributi (Y X Z) tale che

$t \in R \bowtie S \Leftrightarrow t[YX] \in R \wedge t[ZX] \in S.$

Riferendoci al precedente esempio, avremmo un risultato leggermente diverso: il tipo del risultato ha un attributo in meno poichè gli attributi dallo stesso nome sono riportati una volta sola, e non è necessario rinominare essendo tutti gli attributi ora distinti. In generale si preferisce *equi join*, perchè non è detto che due attributi dai nomi uguali siano legati da una corrispondenza chiave primaria - chiave esterna, e nemmeno viceversa che due attributi dai nomi diversi non siano invece in relazione chiave primaria - chiave esterna. Si considerino per esempio le relazioni

Studenti(Matricola, Nome, Provincia, AnnoNascita)
Esami(Codice, Materia, Candidato, Voto, Lode)

Dove l'attributo **Candidato** è chiave esterna per **Matricola**. La giunzione naturale avrebbe come condizione **true**, degenerando di fatto in un prodotto. Mentre *equi join*, ha correttamente la condizione **Matricola** = **Candidato**.

Figura 2.7 Albero per il groupby

2.1.5 Raggruppamento γ

$_{A_1,\ldots,A_n}\gamma_{f_1,\ldots,f_m}(E)$

con $A_1, \ldots, A_n$ attributi di R ed $f_1, \ldots, f_m$ *funzioni di aggregazione* con argomenti che sono espressioni aritmetiche ottenute a partire da costanti e attributi di R. Esempi di funzioni di aggregazione sono la media **AVG**, la somma **SUM**, il conteggio degli elementi **COUNT**, il minimo e il massimo **MIN** e **MAX**. Questi operatori sono noti come *funzioni di aggregazione*.

Il risultato si calcola in questo modo:

- si partizionano le ennuple di R in un insieme di gruppi, mettendo nello stesso gruppo tutte le ennuple che coincidono su tutti gli attributi $A_1, \ldots, A_n$. Se l'insieme $\{A_1, \ldots, A_n\}$ è vuoto, gli elementi di E fanno parte di un unico gruppo;
- si calcolano le funzioni di aggregazione per ogni gruppo;
- si restituisce una relazione con un'ennupla per ogni gruppo e componenti i valori degli attributi $A_1, \ldots, A_n$ e i risultati delle espressioni $f_1, \ldots, f_m$.

Esempio: Gli anni di nascita di tutti i giocatori e il numero di giocatori nati per ciascun anno: BirthYear γ COUNT(*) (Players).

Come nella proiezione generalizzata, gli attributi del risultato si possono rinominare usando l'operatore AS.

Il tipo del risultato è **(BirthYear integer, COUNT(*) integer)**, in accordo con la definizione data. In quanto nodo radice si è omesso di rinominare **COUNT(*)** ma per ovvi motivi va rinominato in caso vi si voglia riferire in quanto attributo contenente il risultato di questa particolare applicazione della funzione **COUNT** e non come una nuova applicazione di funzione. Inoltre se in un'interrogazione si calcolano più volte le funzioni di aggregazione, bisogna per forza rinominare

per evitare confusione. Le funzioni di aggregazione sono molto usate in pratica, e consentono di calcolare semplici dati statistici spesso richiesti.

2.1.6 Proiezione su multinsiemi π^b e ordinamento τ

Tali operatori non sono propriamente operatori algebrici, infatti non operano su insiemi, ma vengono aggiunti perchè nei DBMS sono di uso comune per realizzare funzionalità che altrimenti sarebbero mancanti.

- π^b è analogo alla proiezione ma opera su multiinsiemi poichè non elimina i duplicati. Tale funzionalità può risultare utile per esempio, se si vogliono calcolare funzioni di aggregazione su tutti gli elementi della relazione originaria, anche se proiettata.
- τ ordina la relazione su uno o più attributi, e dal momento che gli insiemi non hanno un ordinamento, anche in questo caso il risultato non è un insieme.

Più formalmente, per questi due operatori, se il tipo in entrata è una relazione e cioè un'insieme finito di ennuple, il tipo del risultato sarà una sequenza finita di ennuple, indicata con seq{}.

2.2 SQL

Nei RDBMS commerciali (e non), le interrogazioni non si formulano in algebra relazionale ma in un linguaggio più semplice da comprendere e usare per l'utente finale: **SQL**. Si tratta di un linguaggio dichiarativo progettato per gestire dati in RDBMS e basto originariamente sull'algebra relazionale. Permette di formulare interrogazioni, aggiornare i dati e creare e modificare gli schemi dei dati, oltre che gestire l'autenticazione e i permessi di accesso ai dati. È stato uno dei primi linguaggi per il modello relazionale proposto da Edgar F. Codd in [3]. Si può suddividere in 3 sottoinsiemi:

- Data Definition Language (**DDL**), per creare e cancellare database o di modificarne la struttura
- Data Manipulation Language (**DML**), per inserire, cancellare, modificare e leggere i dati

– Data Control Language (**DCL**), per di gestire gli utenti e i permessi.

Nel JRS è implementato un sottoinsieme dell'**SQL**, la cui grammatica è riportata nell'helpfile del JRS stesso.

Per motivi pratici, **SQL** apporta alcune estensioni all'algebra, a cui si è già accennato, come l'operatore di ragruppamento o il permettere duplicati all'interno delle relazioni, operando quindi su sequenze. I problemi legati a queste estensioni sono alla base delle critiche rivolte a **SQL** e hanno determinato l'evoluzione dei RDBMS per risolvere tra l'altro, le questioni poste in [4]. Con la definizione di algebra data, le interrogazioni esprimibili nell'algebra relazionale sono un sottoinsieme di quelle esprimibili in **SQL**, per cui data un'espressione in algebra è sempre possibile formulare un'interrogazione in **SQL** equivalente.

2.3 Gli operatori fisici

Gli RDBMS, in presenza di un'interrogazione in **SQL**, generano, tramite l'ottimizzatore, un *piano di accesso*. Data una query, vi sono più piani logici equivalenti, e dato un piano logico, vi sono più piani di accesso equivalenti. Compito dell'ottimizzatore è trovare il miglior piano possibile per eseguire la query data. Similmente a un'interrogazione in algebra relazionale e quindi ad un piano logico, un piano di accesso è costituito da un'insieme di operatori fisici annidati, che però non operano su relazioni, ma su *sequenze di record*. Ogni operatore e quindi ogni piano corrisponde a un algoritmo per l'esecuzione della query di partenza, e realizzano gli operatori algebrici. Tutti gli operatori operano su sequenze di record e restituiscono sequenze di record, e, nel caso del JRS, realizzano tutti l'interfaccia di iteratore. Tale interfaccia consiste in [2]:

– **open** inizializza lo stato dell'operatore ed esegue il metodo open degli operatori fisici dei suoi argomenti;
– **next** ritorna il record successivo del risultato interagendo con gli operatori fisici dei suoi argomenti;

– **isDone** restituisce true se non ci sono altri record da ritornare, false altrimenti;
– **close** termina le operazioni delloperatore fisico e dei suoi argomenti.

Essendo algoritmi, alcuni operatori richiedono che i record siano ordinati, o privi di duplicati, o entrambi. Non solo per ogni piano logico vi sono più piani di accesso equivalenti, ma anche un singolo operatore algebrico può essere realizzato con più operatori fisici. Alcuni operatori fisici fanno uso di *indici*.

Un indice [2] su un attributo A di una relazione R, dal punto di vista logico, è una relazione con due attributi (A, TID), con gli elementi ordinati su A e valori (k_i, r_j), dove k_i è un valore di A presente in un record di R, ed r_j è un riferimento (TID) al record di R in cui A vale k_i. Se A non è una chiave, nell'indice sono presenti tanti record (k_i, r_j) con lo stesso valore k_i di A quanti sono i record di R in cui A vale k_i. Segue la definizione degli operatori fisici, raggruppati secondo l'operatore algebrico che realizzano. Per la forma si fa riferimento al formulario in [2].

2.3.1 Operatori per le tabelle

Intendendo con R una tabella, con Idx un indice su R, con $\{A_i\}$ un insieme di attributi e con O, O_E, O_I operandi, operando esterno e operando interno rispettivamente:

– **TableScan**(R) scansione di R. Trova tutti i record di R, scandendo tutta la tabella dal primo all'ultimo record.
– **IndexScan**(R, Idx) scansione ordinata di R sugli attributi dellindice Idx.
– **SortScan**(R, $\{A_i\}$) scansione di R ordinata sugli $\{A_i\}$.

2.3.2 Operatori per la proiezione

– **Project**(O, $\{A_i\}$) proiezione dei record di O senza leliminazione dei duplicati.
– **Distinct**(O) eliminazione dei duplicati dai record di O ordinati sugli $\{A_i\}$. Richiede che i record di O siano ordinati

2.3.3 Operatori per la restrizione

- **Filter**(O, Φ) restrizione dei record di O: trova tutti i record di O che soddisfano la condizione ϕ
- **IndexFilter**(R, Idx, Φ) restrizione di R con l'indice Idx e accessi a R.
- **IndexOnlyFilter**$(R, Idx, \{A_i\}, \Phi)$ restrizione di R con l'indice Idx senza accessi a R. L'operatore ritorna i record di R che soddisfano Φ, con attributi gli $\{A_i\}$, un sottoinsieme degli attributi sui quali è definito lindice.

Volendo dare degli esempi di piani di accesso che usano gli operatori appena definiti, consideriamo l'albero logico di Figura 2.3 , riportato per comodità in Figura 2.8 . Una traduzione banale è presentata nella Figura 2.8 , seguita da una traduzione più efficiente, dove **PLAYER_Idx** è un indice definito su {**Name**, **BirthYear**}. Il secondo piano di accesso è più efficiente sotto l'ipotesi che **Sort** materializzi il risultato in una tabella temporanea, risultando in accessi casuali in scrittura su disco, mentre **IndexScan** effettua accessi in lettura casuali.

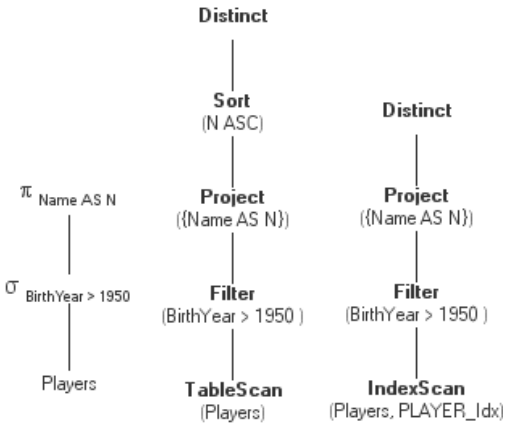

Figura 2.8 Esempio di piano di accesso per l'albero logico di Figura 2.3, e piano di accesso più efficiente

2.3.4 Operatori per la giunzione

- **NestedLoop**(O_E, O_I, Φ_J) giunzione con l'algoritmo *nested loop*.
- **IndexNestedLoop**(O_E, O_I, Φ_J) giunzione con l'algoritmo *index nested loop*; O_I utilizza un indice Idx definito sugli attributi di giunzione, e può essere solo **IndexFilter**(R, Idx, Φ) oppure **Filter**(**IndexFilter**$(R, Idx, \Phi), \psi)$.

– **MergeJoin**(O_E, O_I, Φ_J) giunzione con l'algoritmo *merge join*. I record degli operandi O_E e O_I sono ordinati sugli attributi di giunzione, chiave in O_E e chiave esterna in O_I.

Come esempio si veda il piano logico di Figura 2.6, riportato in Figura 2.9 per comodità. Sono mostrati due possibili piani di accesso, uno con **NestedLoop** e un altro con **IndexNestedLoop**.

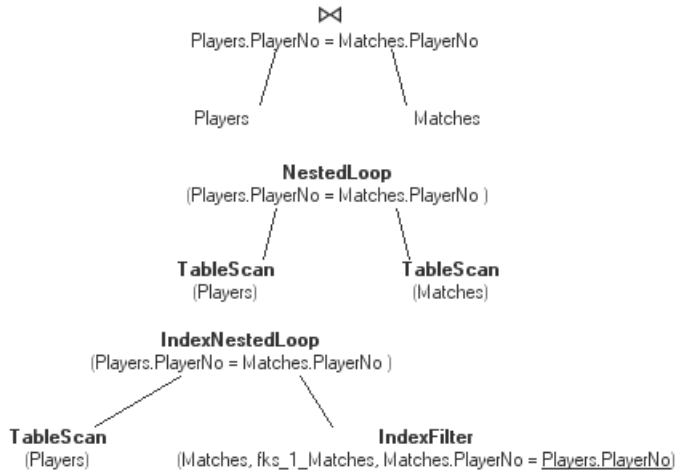

Figura 2.9 Esempio di piano di accesso per l'albero logico di Figura 2.6, e piano di accesso più efficiente

2.3.5 Operatori per l'ordinamento

Sort$(O, \{A_i\})$ ordinamento dei record di O sugli $\{A_i\}$.

2.3.6 Operatori per il raggruppamento

GroupBy$(O, \{A_i\}, \{f_i\})$ raggruppamento dei record di O ordinati sugli $\{A_i\}$ usando le funzioni di aggregazione in $\{f_i\}$. L'operatore ritorna record con attributi gli A_i e le funzioni di aggregazione in $\{f_i\}$, ordinati sugli $\{A_i\}$.

2.3.7 Operatori insiemistici

– **Union**, **Intersect**, **Except**(O_E, O_I) operatori insiemistici con i record degli operandi ordinati e privi di duplicati.

– **UnionAll**(O_E, O_I) unione con duplicati dei record degli operandi.

2.4 Conclusioni

È stata data una concisa definizione dell'algebra relazionale, chiarendo in che modo si estende l'algebra per l'uso pratico. Sono state ricordate alcune caratteristiche dell'**SQL**, e infine sono stati presenati gli operatori fisici del JRS che realizzano gli operatori algebrici definiti. Nel prossimo capitolo, si mostra come creare alberi logici o fisici nell'editore grafico, con nodi che rappresentano gli operatori descritti.

Capitolo 3

L'EDITORE GRAFICO DI ALBERI LOGICI E FISICI

Si descrivono le funzionalità dell'editore grafico per definire ed eseguire su una base di dati interrogazioni formulate con un albero logico dell'algebra relazionale o con un albero fisico che descrive un algoritmo per eseguire l'interrogazione usando gli operatori fisici del DBMS relazionale JRS (*Java Relational System*), sviluppato in Java per scopi didattici presso il Dipartimento di Informatica dell'Università di Pisa.

3.1 Introduzione

I due editori grafici per alberi logici e fisici (detti anche *piani*) si attivano dal JRS con i pulsanti **Logical Plan** e **Physical Plan** ed hanno un'interfaccia simile per quanto riguarda il modo in cui si definiscono i nodi di un albero, gli archi fra i nodi e come si opera su un albero. Le differenze riguardano i tipi nodi disponibili e il modo diverso di eseguire i piani. Si presentano prima le caratteristiche simili e poi quelle specifiche per tipo di albero.

L'attivazione di un editore grafico visualizza una nuova finestra divisa in tre aree principali:

– Un'area per definire un albero (**Logical Plan** o **Physical Plan**).

– Un'area che contiene il risultato dell'esecuzione di un piano, che dipende dal tipo di albero (**Query** o **Output**).

– Un'area **Control Panel** che contiene i pulsanti per aggiungere un nodo, rimuovere un nodo, salvare, caricare o cancellare un piano.

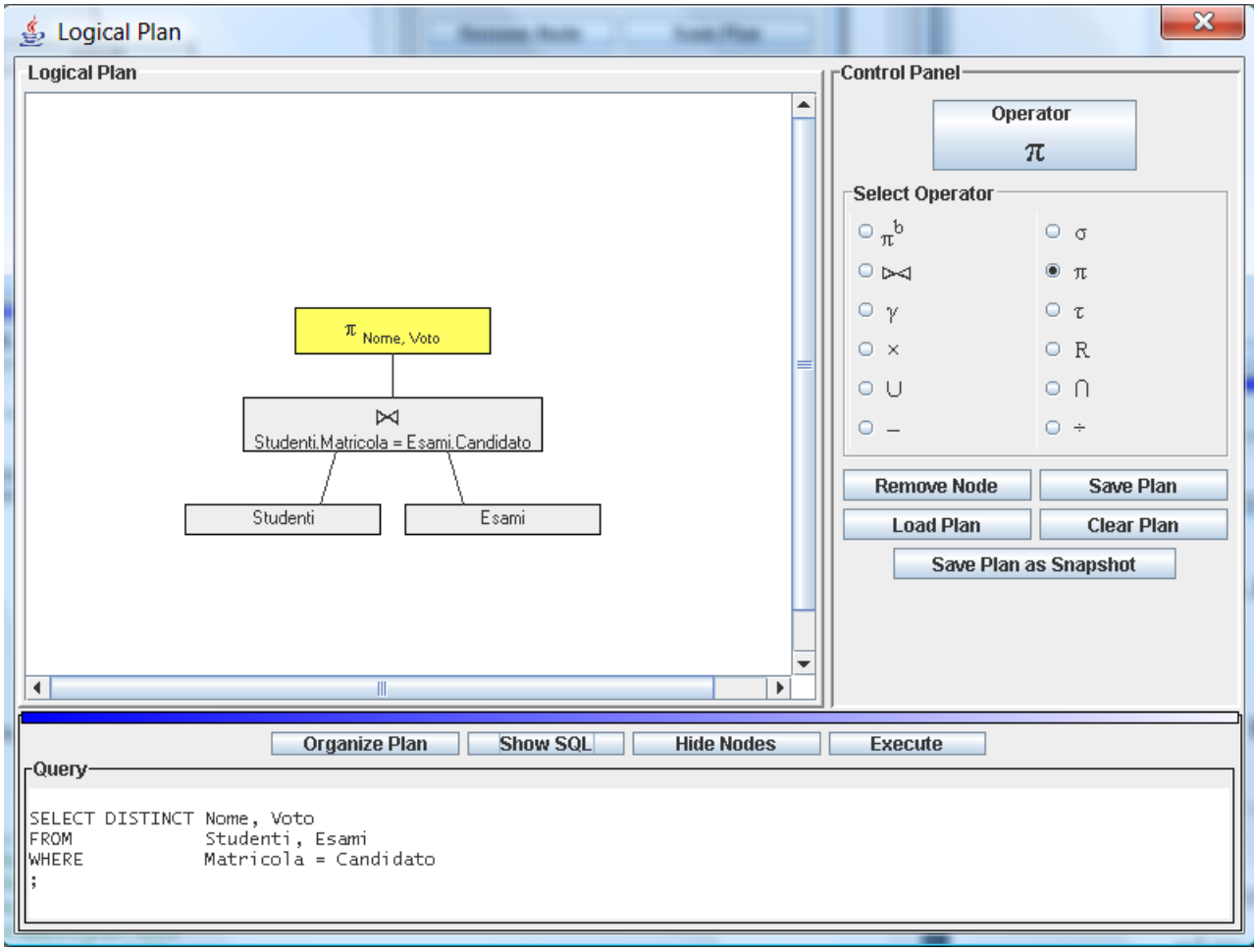

Figura 3.1 L'editor di piani logici

3.1.1 Il Control Panel

L'area è suddivisa in tre parti:

– Un riquadro **Select Operator** che elenca gli operatori che si possono aggiungere ad un albero, selezionabili con un clic del mouse. La selezione di un operatore lo fa apparire nell'area degli alberi.

– Un pulsante **Operator** che mostra l'operatore selezionato. Per aggiungere altri operatori dello stesso tipo nell'area dell'albero, si fa un clic sul pulsante.
– Un'area con cinque pulsanti per operazioni sui nodi e sugli alberi.
  – **Remove Node** per rimuove un nodo selezionato con un clic e mostrato in giallo.
  – **Save Plan** e **Load Plan** per salvare o caricare un piano. Una volta premuto uno dei due pulsanti, appare una finestra di dialogo per completare l'operazione scelta. Poiché ciascun piano è relativo a una base di dati presente nel JRS, l'operazione di caricamento di un piano viene completata se a base di dati correntemente in uso coincida con quella prevista piano scelto. In caso contrario viene segnalato di selezionare prima la base di dati usata dal piano da caricare.
  – **Clear Plan** cancella il contenuto dell'area degli alberi, se l'operazione viene confermata.
  – **Save Plan as Snapshot** permette di salvare in un file una foto (*snapshot*) del piano in formato jpg.

3.1.2 L'area degli alberi e le operazioni sui singoli nodi

Una volta aggiunto un nodo, questo appare nell'area destinata a contenere l'albero relativo al piano che si sta creando, denominata **Logical Plan** o **Physical Plan**. Su un nodo sono previste le seguenti operazioni:

– *Selezione* con un singolo clic del mouse.
– *Spostamento* trascinandolo con il puntatore del mouse, tenendo premuto il pulsante sinistro. Per spostare un intero sottoalbero, si esegue la stessa operazione sulla radice, ma tenendo premuto il tasto **Maiuscola**.
– *Aggiunta di un arco* fra due nodi. Quando il puntatore del mouse è su un nodo, sul suo bordo appaiono dei quadratini *maniglie*. Per aggiungere un arco, bisogna muovere il puntatore sopra una maniglia del nodo di partenza, tenere premuto il tasto sinistro del mouse e spostarlo verso il nodo di arrivo, dove si rilascia il pulsante.

– *Rimozione di un arco* spostandone un'estremità da una maniglia in un'area priva di nodi.
– *Specifica di un nodo* con un *menu contestuale* che si visualizza cliccando con il pulsante destro del mouse sul nodo. Tale menu è diverso per ogni nodo, e consente di inserire i parametri. L'inserimento dei parametri è quindi guidato attraverso i menu contestuali, che consentono di specificare solo i parametri previsti dal tipo di nodo e dai suoi operandi, che nell'albero sono i figli.

3.1.3 L'area dei risultati

L'area è detta **Query** nel caso dei piani logici e **Output** nel caso dei piani fisici. Nell'area viene stampato il risultato della valutazione di un albero o di un sottoalbero con radice il nodo selezionato.

Al di sopra dell'area sono presenti i seguenti pulsanti e una barra blu che serve a ridimensionare con il mouse l'area dei risultati.

Hide/Show Nodes

Questo pulsante permette di nascondere i rettangoli dei nodi, per poter visualizzare l'albero come solo testo e archi. Una volta premuto il bottone, l'interfaccia grafica relativa ai nodi viene nascosta e l'etichetta del pulsante cambia in **Show Nodes** per rendere visibile l'interfaccia grafica dei nodi.

Execute Plan

Questo pulsante permette di eseguire il piano con *radice il nodo selezionato*. Questa funzionalità consente di vedere la valutazione di un piano gradualmente a partire dalle foglie. Il risultato viene stampato nella finestra di output e mostrato in un'apposita finestra prevista dal JRS, in modo formattato e ridimensionabile.

Organize Plan

Questo pulsante consente ridisegnare automaticamente un albero senza nodi sovrapposti e ben allineati. I nodi con un solo figlio, avranno il nodo figlio immediatamente al di sotto, allineato al centro, in modo che l'arco sia verticale. Se hanno due figli, questi vengono posizionati in modo da evitare sovrapposizioni.

E' possibile spostare l'albero ridisegnato o un sottoalbero con il mouse tenendo premuto il tasto **Maiuscola**.

Show SQL

Nell'editor di piani logici è possibile vedere nell'area **Query** la traduzione dell'albero logico nel codice SQL usato per eseguirlo.

3.2 I menu contestuali dell'editore di alberi logici

Per ciascun nodo, il menu contestuale è diverso, viene creato dinamicamente al momento dell'inserimento del nodo nell'albero e tiene conto degli attributi degli operandi e dello schema della base di dati. Ogni scelta dei parametri comporta un aggiornamento dell'etichetta del nodo per rispecchiare le scelte fatte. Nel creare un albero, per via delle dipendenze degli operatori, conviene procedere dal basso verso l'alto con l'immissione dei parametri. I menu contestuali diventano attivi solo quando un nodo ha tutti gli archi uscenti che richiede. Segue una breve descrizione di ciascun menu.

R

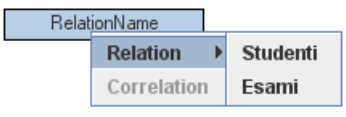

Figura 3.2 Menu del nodo Relation

Il menu ha le seguenti parti:

– **Relation** per la scelta della relazione, che elenca tutte le tabelle definite nella base di dati corrente, *che devono essere definite con almeno una chiave*;
– **Correlation** per poter dare il nome della variabile di correlazione. Questo comando è disabilitato finchè non si è scelto una tabella nel sottomenu **Relation**. Il modo corretto di procedere consiste quindi nello scegliere una tabella e poi eventualmente immettere la variabile di correlazione.

π

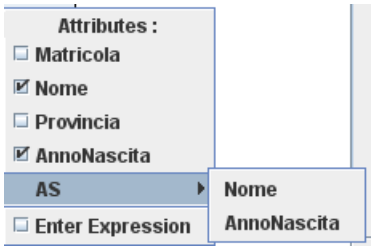

Figura 3.3 Menu del nodo per la proiezione

Il menu ha le seguenti parti:

– **Attributes** per scegliere gli attributi della proiezione, tra quelli presenti nel tipo dell'operando. Tutti gli attributi selezionati, saranno presenti nel sottomenu **AS** che permette di assegnare loro un nuovo nome.
– **Enter expression** per scrivere un'espressione come parametro dell'operatore, che poi appare tra le possibili scelte nel sottomenu **AS** per assegnarle un nome.

σ

Il menu ha le seguenti parti:

– **Condition**, con tre sottomenu l'immissione di un semplice predicato composto da un attributo, un operatore di confronto e un secondo attributo. Nei menu **First Parameter** e **Second Parameter** è inoltre presente un comando **Enter Expression** per scrivere costanti o espressioni.
– **Enter condition** consente di scrivere una condizione complessa.

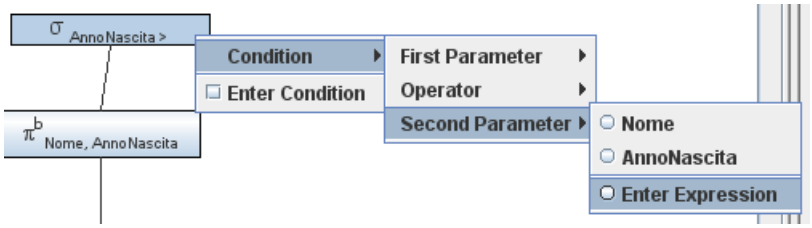

Figura 3.4 Menu del nodo per la restrizione

γ

Il menu ha le seguenti parti:

- **Groupings** per la la scelta degli attributi di raggruppamento.
- **Aggregations** per la scelta delle funzioni di aggregazione, con o senza DISTINCT, applicabili solo agli attributi per le quali sono correttamente definite. Quindi funzioni come SUM e AVG hanno solo attributi di tipo numerico, e l'asterisco è presente soltanto come possibile scelta come argomento del COUNT. Le funzioni di aggregazione sono quelle previste dal JRS. È possibile inserire espressioni come argomenti delle funzioni di aggregazione tramite l'opzione Enter Expression dei sottomenu relativi. *Ogni funzione di aggregazione deve essere rinominata con l'operatore* **AS**.
- **AS** per assegnare un nome agli attributi di raggruppamento.

$\times$

Questo nodo è privo del menu contestuale, poichè non ha parametri. Bisogna semplicemente collegarlo a due sottoalberi con risultato privo di attributi uguali. Se i due sottoalberi non hanno attributi diversi viene dato messaggio di errore.

$\bowtie$

Il menu ha le seguenti parti:

- Le parti previste dal menu σ per specificare una condizione di giunzione generale.

– **Natural Join** per la definizione automatica della condizione di giunzione naturale, considerando gli attributi dei due operandi di uguale nome.

– **Equi Join** per la definizione automatica della condizione di giunzione fra gli attributi della chiave primaria e della chiave esterna corrispondente dei due operandi. Nella Figura 3.5 vediamo un semplice esempio di equi-join:l'attributo *Matricola*, della chiave primaria della relazione *Studenti*, è in relazione con l'attributo *Candidato*, chiave esterna di *Esami* con riferimento a *Studenti*. Attivando il comando Equi Join viene quindi generata la condizione S.Matricola = E.Candidato.

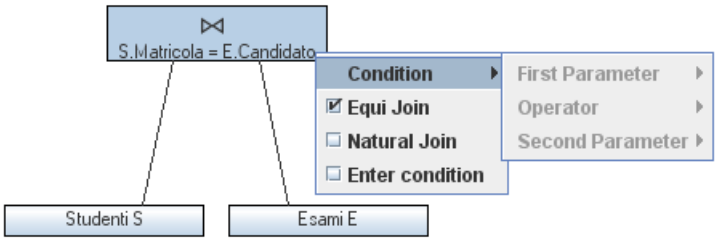

Figura 3.5 Menu del nodo per la giunzione

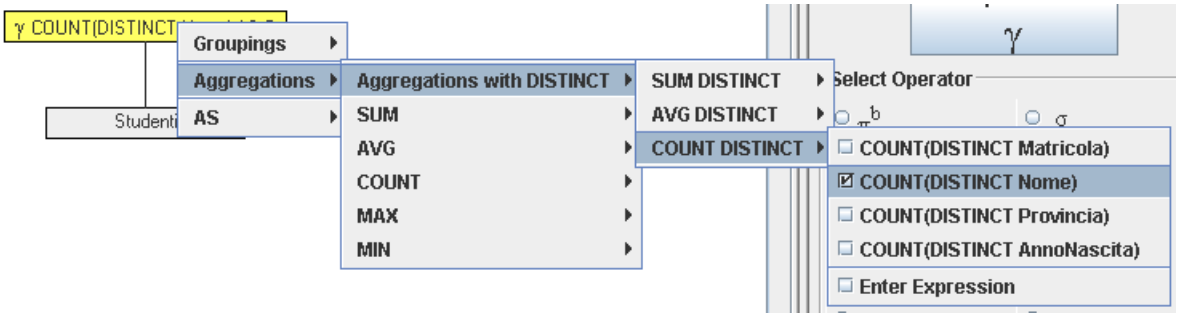

Figura 3.6 Menu del nodo per il raggruppamento

∪, ∩, −

Questi nodi sono privi del menu contestuale, poichè non hanno parametri. Bisogna semplicemente collegarli a due sottoalberi con risultato dello stesso tipo, ovvero gli stessi attributi, con lo stesso tipo primitivo, nello stesso ordine. Se i due sottoalberi non hanno lo stesso tipo viene dato messaggio di errore.

I nodi di un albero logico costruito con i precedenti operatori dell'algebra relazionale hanno come risultato un insieme di record, se le relazioni delle foglie dell'albero, definite in SQL nel JRS, prevedono almeno una chiave. Invece i seguenti operatori hanno in generale come risultato una sequenza di record e sono stati aggiunti per produrre interrogazioni comuni nell'SQL.

π^b

Il menu ha stesse le parti dell'operatore π e in generale produce un risultato con elementi duplicati (**Multiset Projection**).

τ

Il menu ha le seguenti parti:

- **Attributes** per scegliere gli attributi di ordinamento.
- **Direction** per scegliere l'ordinamento ascendente o decrescente.

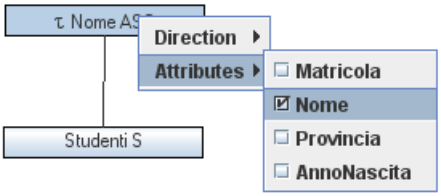

Figura 3.7 Menu del nodo per l'ordinamento

L'editore grafico consente l'uso di π^b o τ solo come radice di un albero logico. Se si usano entrambi, la radice deve essere τ.

3.2.1 La visualizzazione delle informazioni relative ad un nodo

Le informazioni relative a un nodo si visualizzano con un doppio clic su di esso. Le informazioni dipendono dal tipo di nodo:

- **Operator** l'operatore di cui si stanno visualizzando le informazioni;
- **Relation** la tabella usata, se è un nodo foglia;
- **Condition** per i nodi che prevedono una condizione come parametro;

– **Result Type** il tipo del risultato del nodo logico, rappresentato nella forma $\{(A_1\ T_1, \ldots, A_n\ T_n)\}$, se è un insieme, o nella forma **seq**$\{(A_1\ T_1, \ldots, A_n\ T_n)\}$, se è un multiinsieme o un insieme ordinato.

3.3 I menu contestuali dell'editore di alberi fisici

Per ciascun nodo, il menu contestuale è diverso, viene creato dinamicamente al momento dell'inserimento del nodo nell'albero, e tiene conto degli attributi degli operandi e dello schema della base di dati. Ogni scelta dei parametri comporta un aggiornamento dell'etichetta del nodo per rispecchiare le scelte fatte. Nel creare un albero, per via delle dipendenze degli operatori, conviene procedere dal basso verso l'alto con l'immissione dei parametri. I menu contestuali diventano attivi solo quando un nodo ha tutti gli archi uscenti che richiede. Segue una breve descrizione di ciascun menu.

3.3.1 Operatori su tabelle

TableScan

Il menu ha le seguenti parti:

– **Tables** per scegliere la tabella tra quelle definite nella base di dati corrente.
– **Correlation** per specificare una variabile di correlazione.

IndexScan

Il menu ha le seguenti parti:

– Le parti previste dal menu di **TableScan**.
– **Index** per la la scelta dell'indice, tra quelli definiti nella base di dati corrente sulla tabella scelta. Questo menu è disabilitato finchè non si sceglie una tabella.

SortScan

Il menu ha le seguenti parti:

– Le parti previste dal menu di **TableScan**.

– Le parti previste dal menu di **Sort**.

3.3.2 Operatori per l'ordinamento

Sort

Il menu ha le seguenti parti:

– **Attributes** per scegliere gli attributi di ordinamento. Gli attributi vengono aggiunti all'inizio dei parametri del nodo, quindi se per esempio volessimo ordinare per A e B va selezionato prima B e poi A.

– **Direction** per scegliere l'ordinamento ascendente o decrescente.

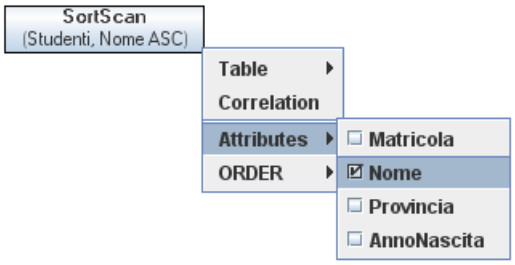

Figura 3.8 Menu del nodo SortScan

3.3.3 Operatori per la proiezione

Project

Il menu ha le seguenti parti:

– **Attributes** per scegliere gli attributi della proiezione, tra quelli presenti nel tipo dell'operando. Tutti gli attributi selezionati, saranno presenti nel sottomenu **AS** che permette di assegnare loro un nuovo nome.

– **Enter expression** per scrivere un'espressione come parametro dell'operatore, che poi appare tra le possibili scelte nel sottomenu **AS** per assegnarle un nome.

L'operatore fisico **Project** è equivalente all'operatore **Multiset Projection** e non all'operatore **Projection** dell'algebra relazionale, perché non elimina eventuali duplicati dal risultato, che invece si eliminano con il prossimo operatore **Distinct**.

Distinct

Il nodo non ha un menu contestuale, poichè non ha parametri. Elimina semplicemente i duplicati dai record ordinati dell'operando. Si noti che per il corretto funzionamento dell'operatore i dati dell'operando devono essere ordinati su tutti i suoi attributi.

3.3.4 Operatori per la restrizione

Filter

Il menu ha le seguenti parti:

- **Condition**, con tre sottomenu l'immissione di un semplice predicato composto da un attributo, un operatore di confronto e un secondo attributo. Nei menu **First Parameter** e **Second Parameter** è inoltre presente un comando **Enter Expression** per scrivere costanti o espressioni.
- **Enter condition** consente di scrivere una condizione complessa.

IndexFilter

Il menu ha le seguenti parti:

- **Table** e **Correlation** per scegliere una tabella, sulla quale è definito un indice, e definire un'eventuale variabile di correlazione come in **TableScan**.
- **Index** per scegliere un indice tra quelli definiti sulla tabella selezionata. Gli attributi dell'indice saranno poi gli unici visibili nel menu **Condition** .
- **Condition** e **Enter Condition** per definire un predicato sugli attributi dell'indice. Il sottomenu **Attribute** consente di scegliere un attributo dell'indice scelto, mentre il secondo **Value** oltre agli attributi dell'indice presenta anche

l'opzione **Enter Expression** il cui uso è per le costanti degli operatori di confronto. Per condizioni generali si utilizza **Enter Condition**.

– **Join Condition** ψ per specificare la condizione di giunzione quando l'operatore è usato come operando interno di un nodo **IndexNestedLoop**. Quando si collega con un arco al nodo **IndexNestedLoop**, nell'etichetta del nodo comparirà la condizione di giunzione come parametro, con l'attributo appartenente all'operando esterno sottolineato. Quando si usa un **IndexNestedLoop** si raccomanda di completare prima i parametri di **IndexFilter**.

Il menu presenta i parametri nell'ordine in cui vanno immessi: prima la tabella, poi eventualmente la variabile di correlazione, l'indice e infine la condizione.

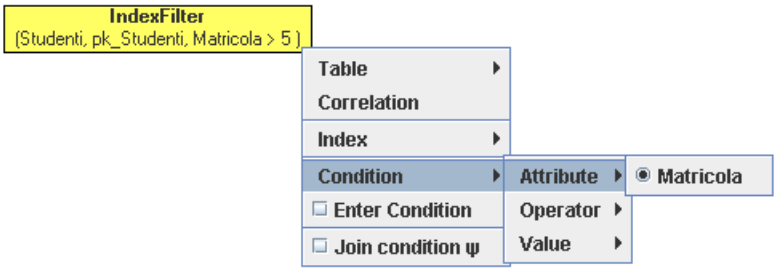

Figura 3.9 Menu del nodo IndexFilter

IndexOnlyFilter

Il menu ha le seguenti parti:

– **Index**, **Condition**, **Enter Condition**, **Join Condition** ψ identiche al menu di **IndexFilter**.

– **Project** per scegliere quali attributi dell'indice devono far parte del risultato.

3.3.5 Operatori per il raggruppamento

GroupBy

Il menu ha le seguenti parti:

– **Groupings** per la la scelta degli attributi di raggruppamento.

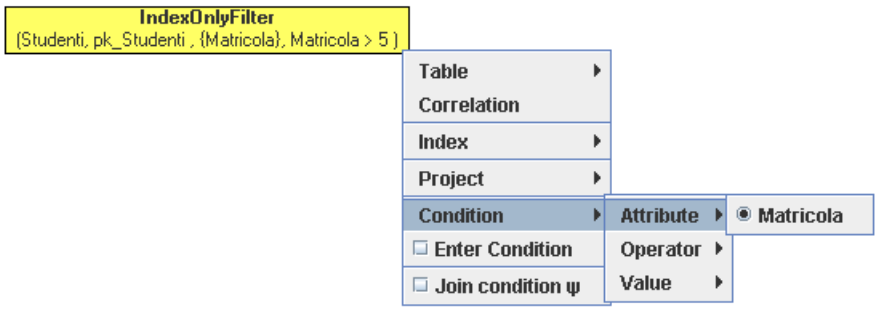

Figura 3.10 Menu del nodo IndexOnlyFilter

- **Aggregations** per la scelta delle funzioni di aggregazione, con o senza DISTINCT, applicabili solo agli attributi per le quali sono correttamente definite. Quindi funzioni come SUM e AVG hanno solo attributi di tipo numerico, e l'asterisco è presente soltanto come possibile scelta come argomento del COUNT. Le funzioni di aggregazione sono quelle previste dal JRS. *Ogni funzione di aggregazione deve essere rinominata con* **AS**.
- **AS** per assegnare un nome agli attributi di raggruppamento.

Si noti che per il corretto funzionamento dell'operatore i dati dell'operando devono essere ordinati sugli attributi di raggruppamento.

3.3.6 Operatori per la giunzione

NestedLoop

Il menu ha le seguenti parti:

- **Condition**, con tre sottomenu per specificare un predicato composto da un attributo, un operatore di confronto e un secondo attributo. Nei menu **First Parameter** e **Second Parameter** è inoltre presente un comando **Enter Expression** per scrivere costanti o espressioni.
- **Enter condition** per specificare una condizione generale.
- **Equi Join** per la definizione automatica della condizione di giunzione fra gli attributi della chiave primaria e della chiave esterna corrispondente dei due operandi, se sono definite.

IndexNestedLoop

Il menu è analogo a quello di **NestedLoop**, con il vincolo che l'operando interno deve essere un nodo **IndexFilter** o **IndexOnlyFilter**, oppure un nodo **Filter** seguito da **IndexFilter** o **IndexOnlyFilter**.

Figura 3.11 Esempio di IndexNestedLoop associata a IndexFilter

MergeJoin

Il menu è analogo a quello di **NestedLoop**, con il vincolo che i record degli operandi sono ordinati sugli attributi di giunzione, chiave nell'operando esterno e chiave esterna nell'operando interno.

3.3.7 Operatori insiemistici

Gli operatori insiemistici non hanno un menu contestuale. Si controlla che i tipi degli operandi siano gli stessi.

3.3.8 La visualizzazione delle informazioni sui nodi

Le informazioni relative a un nodo si visualizzano con un doppio clic su di esso. Le informazioni dipendono dal tipo del nodo:

- **Operator** l'operatore di cui si stanno visualizzando le informazioni;
- **Table** la tabella usata, se è un nodo foglia;
- **Index** il nome dell'indice, se è un nodo di un operatore che lo prevede;
- **Attributes** gli attributi dell'indice usato;
- **Condition** la codizione, per i nodi che la prevedono;
- **Tuple Type** il tipo del record del risultato del nodo, nella forma (A_1 T_1, ..., A_n T_n).

– **Order** se l'operatore produce dati ordinati su certi attributi.

Nella Figura 3.12 è mostrato un esempio della finestra **Node Info**.

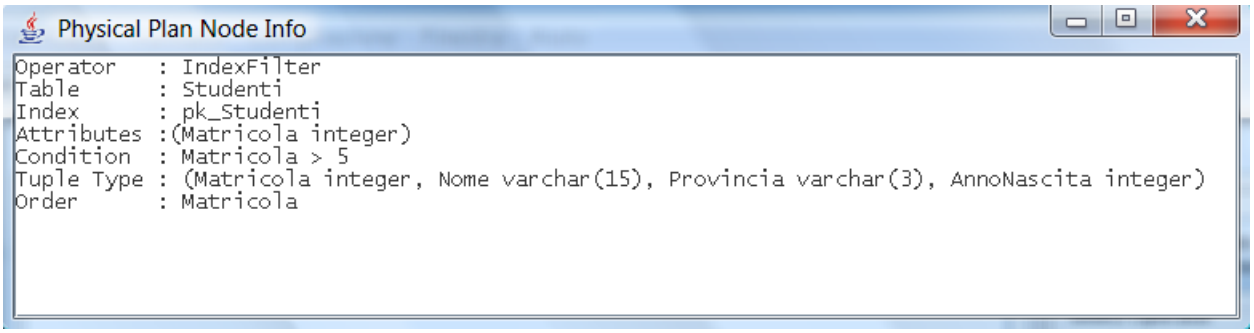

Figura 3.12 Esempio di node info

3.4 Conclusioni

Sono state presentate le funzionalità dell'editore grafico per definire ed eseguire su una base di dati interrogazioni formulate con un albero logico dell'algebra relazionale o con un albero fisico per eseguire l'interrogazione usando gli operatori fisici del DBMS relazionale JRS (*Java Relational System*). Nel prossimo capitolo si descrive un caso d'uso del sistema, allo scopo di presentare un esempio per meglio mostrare le funzionalità e le finalità del progetto.

Capitolo 4

UN CASO D'USO DEL SISTEMA

Si mostra un esempio di utilizzo del sistema, usando sia i piani logici che fisici e confrontando i piani proposti con quelli generati dall'ottimizzatore del JRS.

4.1 La base di dati di esempio

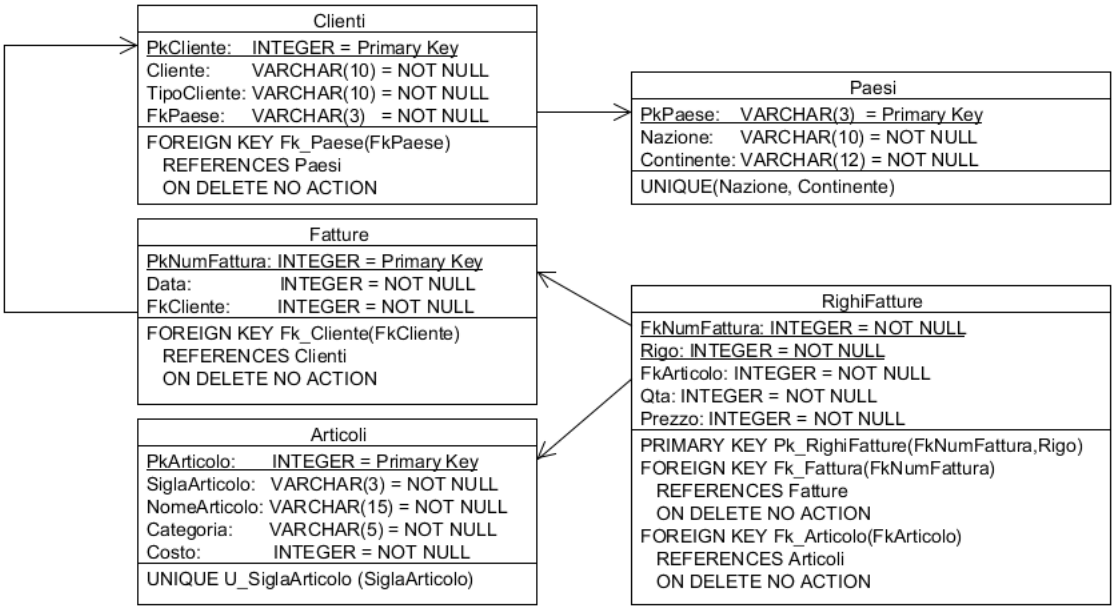

Figura 4.1 Schema della base di dati usata per l'esempio

Lo schema di Figura 4.1 va interpretato come segue:

- Le tabelle sono dei rettangoli, in cima ai quali c'è il nome della tabella.
- nel secondo riquadro sono elencati gli attributi, nella forma attributo: tipo = vincolo
- nel terzo riquadro sono elencati i vincoli d'integrità. Si noti che il vincolo di chiave primaria viene specificato nel terzo riquadro solo in presenza di chiave primaria definita su più attributi.
- per ogni chiave esterna c'è una freccia che parte dalla chiave esterna e termina nella chiave primaria cui fa riferimento.

Lo schema è quindi costituito da 5 tabelle, di cui 4 hanno vincoli di chiave esterna, e tutte hanno una chiave primaria. La tabella **RighiFatture** è in relazione molti-a-uno con la tabella **Fatture**, e uno-a-uno con la tabella **Articoli**.

4.2 L'interrogazione di esempio

Si consideri la seguente interrogazione **SQL**:

```
SELECT    FkArticolo, SUM(Qta) AS SQta
FROM      RighiFatture, Articoli
WHERE     FkArticolo = PkArticolo AND Qta = 1
GROUP BY FkArticolo
HAVING    COUNT(*) > 1;
```

che trova tutti gli articoli venduti in quantità esattamente uguale a uno, ma più volte, e quante volte sono stati venduti in totale. Tale interrogazione è interessante per la presenza di una giunzione unita a un raggruppamento, dove si raggruppa su un attributo di giunzione e la funzione di aggregazione usa un attributo presente in tutto il sottoalbero sinistro della giunzione. Eseguendola in JRS, con l'opzione di mostrare il piano di accesso generato dall'ottimizzatore, si ottiene il piano di Figura 4.2: L'operatore **GroupBy** viene usato dopo la giunzione, come di solito accade in ogni DBMS relazionale.

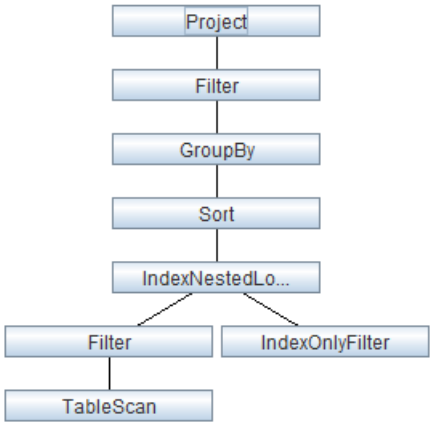

Figura 4.2 Piano di accesso generato dal JRS

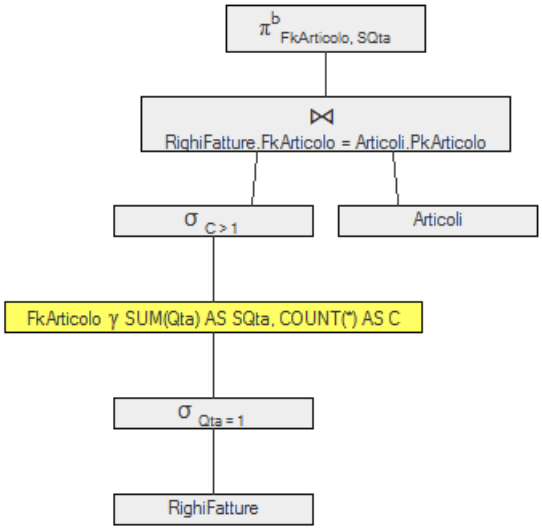

Figura 4.3 Piano logico

4.3 Il piano logico dell'esempio

Si decide di eseguire l'interrogazione con un albero logico in cui si anticipa il **GroupBy** rispetto alla giunzione. Questo è possibile perché vale il seguente risultato [1]:

Proposizione 4.1 Sia $\alpha(X)$ l'insieme degli attributi in X e $R \overset{\bowtie}{_{C_j}} S$ sia una equigiunzione ($C_j = (f_k = p_k)$), con f_k la chiave esterna di R e p_k la chiave primaria di S. R ha la proprietà del *raggruppamento invariante*

$$ {}_A\gamma_F(R \overset{\bowtie}{_{C_j}} S) \equiv \pi^b_{A \cup F}((_{A \cup \alpha(C_j) - \alpha(S)}\gamma_F(R)) \overset{\bowtie}{_{C_j}} S) \tag{4.1} $$

se valgono le seguenti condizioni:

1. $A \rightarrow f_k$, con A gli attributi di raggruppamento della relazione $R \overset{\bowtie}{_{C_j}} S$ e f_k l'attributo di giunzione di R;
2. gli attributi di aggregazione in F sono attributi di R;

Se ci sono restrizioni su R, il γ si anticipa rispetto alla giunzione sulla σ.

In generale, l'anticipazione su tabelle di grandi dimensioni produce piani più efficienti, poichè la giunzione viene fatta su relazioni di cardinalità minore (Figura 4.3) .

L'operatore di raggruppamento è evidenziato, ed è possibile per ogni nodo vedere il tipo del risultato con la finestra **Node Info**. Inoltre si può generare l'interrogazione **SQL** per ogni sottoalbero, oltre che generare l'interrogazione per l'intero albero ed eseguire il piano, con lo stesso risultato ottenuto eseguendo l'interrogazione di esempio vista nella sezione precedente. Per eseguire un piano logico, l'editor genera l'interrogazione in **SQL**, e per interrogazioni complesse usa delle viste. Per una descrizione dettagliata si veda il Capitolo 5. L'interrogazione generata è:

```
CREATE VIEW LTV1 AS
SELECT    FkArticolo, SUM(Qta) AS SQta, COUNT(*) AS C
FROM      RighiFatture
WHERE     Qta = 1
GROUP BY  FkArticolo
HAVING    COUNT(*) > 1
;
```

```
SELECT    FkArticolo, SUM(Qta) AS SQta
FROM      LTV1, Articoli
WHERE     FkArticolo = PkArticolo
;
```

In questo caso, l'interrogazione generata usa una vista per l'operando sinistro della giunzione, costringendo l'ottimizzatore a generare due piani distinti, uno per la vista e uno per l'interrogazione.

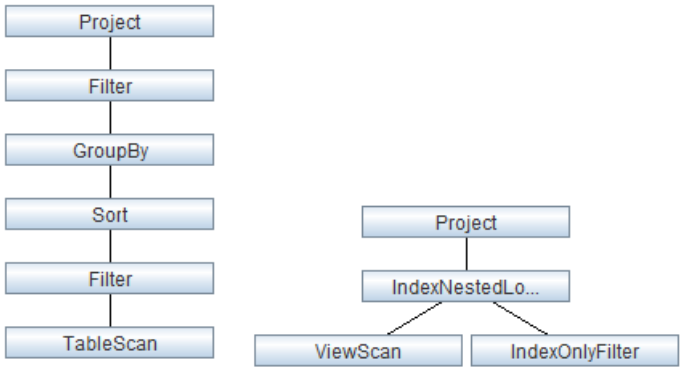

Figura 4.4 Piano di accesso per l'interrogazione generata con viste

4.4 Il piano fisico dell'esempio

Passando all'editor di piani fisici, è possibile disegnare un piano di accesso anticpando il raggruppamento, e realizzare la giunzione utilizzando l'indice sulla chiave primaria di **Articoli**. Si nota immediatamente che tale piano è la combinazione dei piani di Figura 4.4, senza però generare viste. Il calcolo della stima dei costi non viene effettuato nell'editor, ma poichè si conosce tale stima per il piano con vista, è banale concludere che questo piano ha la una stima dei costi più bassa poichè non utilizza una vista e anticipa una proiezione. Anche in questo caso è possibile eseguire dei sottoalberi e vedere le informazioni per ciascun nodo.

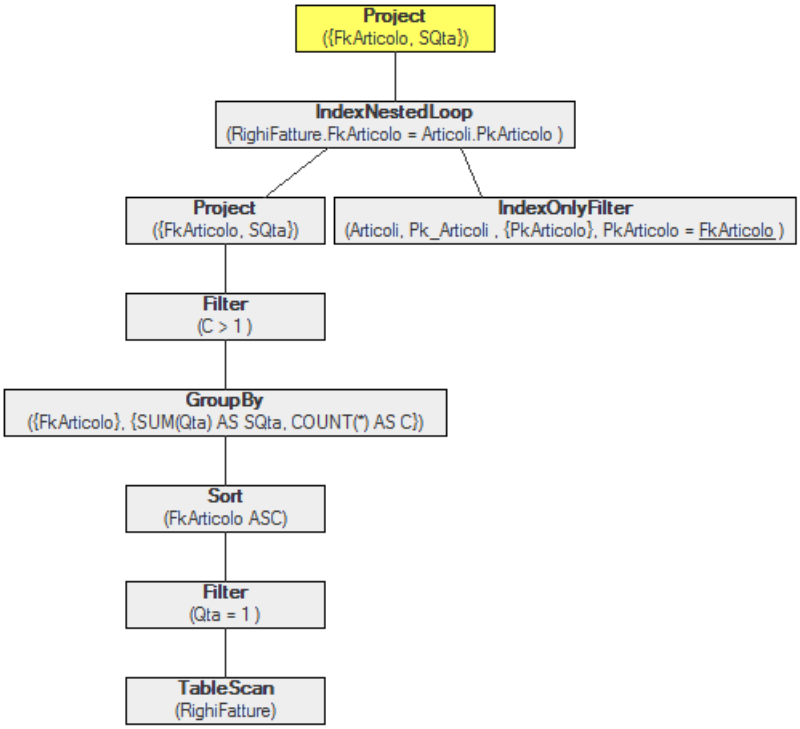

Figura 4.5 Piano fisico per l'esempio

4.5 Conclusioni

È stato mostrato come nel caso di interrogazioni non banali, con l'editore grafico si possano formulare le interrogazioni in modo diverso in algebra relazionale o con dei piani di accesso equivalenti. Con basi di dati con relazioni non piccole si puo' anche vedere come si possano produrre anche piani più efficienti di quelli generati dall'ottimizzatore se non prevede tecniche tipo l'anticipazione dell'operatore di raggruppamento. Nel prossimo capitolo viene descritta l'implementazione del sistema, fornendo uno schema delle classi che guiderà tale descrizione.

Capitolo 5

IMPLEMENTAZIONE DELL'EDITOR

L'implementazione dell'editor ha richiesto un sostanziale lavoro ed è risultato in un'applicazione abbastanza complessa e lunga. Pertanto si presenta prima una visione generale attraverso uno schema delle classi, seguita dalla descrizione dei metodi ritenuti più interessanti.

5.1 Introduzione a Swing

Swing è un'infrastruttura (in inglese *framework*) per Java orientato allo sviluppo di interfacce grafiche. Parte delle classi dell'infrastruttura *Swing* sono implementazioni di oggetti grafici(in inglese *widget*) come caselle di testo, pulsanti, pannelli, tabelle e menu.

Per *look and feel* si intende lo stile di un'interfaccia grafica, i colori, i tasti, le forme dei pulsanti e così via. *Swing* supporta il *look and feel* non tramite quello che viene fornito nativamente dal gestore delle finestre, ma tramite una sua emulazione. Questo significa che si può ottenere un qualsiasi *L&F* supportato su qualsiasi piattaforma. Lo svantaggio di questi componenti è quello di una più lenta esecuzione. Il vantaggio è una uniformità di visualizzazione tra svariate piattaforme. Sui moderni sistemi la lentezza di esecuzione non si nota,

e l'uniformità è a volte troppo rigorosa : per esempio gli utenti di sistemi di tipo *Macintosh*, abituati a altre combinazioni di tasti per eseguire operazioni di base come il *cut&paste*, di fronte a un'applicazione java che è completamente indipendente dal sistema operativo, si trovano dei tasti differenti. Nel JRS si è ovviati a questo problema modificando il *Look&Feel* in modo che nel caso sia eseguito su un sistema del tipo *Macintosh*, imposti le combinazioni in modo analogo a quanto avviene in quei sistemi. Ciò è stato possibile grazie alla modificabilità di *Swing*.

Si tratta quindi una libreria indipendente dalla piattaforma, estendibile, modificabile, configurabile, a spedizione di eventi asincroni, con un unico thread. L'estendibilità è data dal meccanismo delle sottoclassi, utilizzato anche nella realizzazione del'editor : i nodi sono sottoclassi di **JToggleButton**, l'area contenente l'albero e cioè i nodi e gli archi è una sottoclasse di **JPanel**. La modificabilità si realizza con il meccanismo di *override* che permette di riscrivere i metodi in modo opportuno. L'indipendenza dalla piattaforma è data non solo dal fatto che è una libreria Java, ma anche che a differenza di quanto avveniva per le *AWT*, ogni componente non si appoggia sul suo analogo del sistema operativo per la visualizzazione ma è responsabile per la sua stessa visualizzazione. Avere un thread unico riduce il carico di lavoro e rende le applicazioni più leggere. Il trattamento asincrono degli eventi evita di dover attendere inutilmente durante l'esecuzione di un programma, ma ovviamente nei casi in cui c'è necessità di attendere, bisogna farlo in modo esplicito. Tutte queste caratteristiche rendono la scelta di *Swing* quasi obbligata quando si vuole realizzare un'interfaccia grafica in Java.

5.2 Schema delle classi

Lo schema delle classi consiste in un diagramma **UML**, diviso in 3 parti:

– Uno schema dell'editor di piani logici, dove si mostrano le classi utilizzate per realizzare l'interfaccia e i principali metodi dell'editor. Lo schema dell'editor di piani fisici è del tutto simile, pertanto non viene mostrato esplicitamente ma se ne discutono solo alcuni metodi.

– Uno schema dei nodi logici, dove si evidenziano le classi corrispondenti a ogni operatore logico e le classi ed i metodi utilizzati per realizzare l'interfaccia che permette di disegnare alberi e di immettere parametri nei singoli nodi.
– Uno schema dei nodi fisici, dove oltre alle classi e ai metodi descritti nel punto precedente si evidenziano i metodi che associano il nodo dell'albero fisico all'operatore fisico.

5.2.1 Diagramma UML delle classi relative all'editor di piani logici

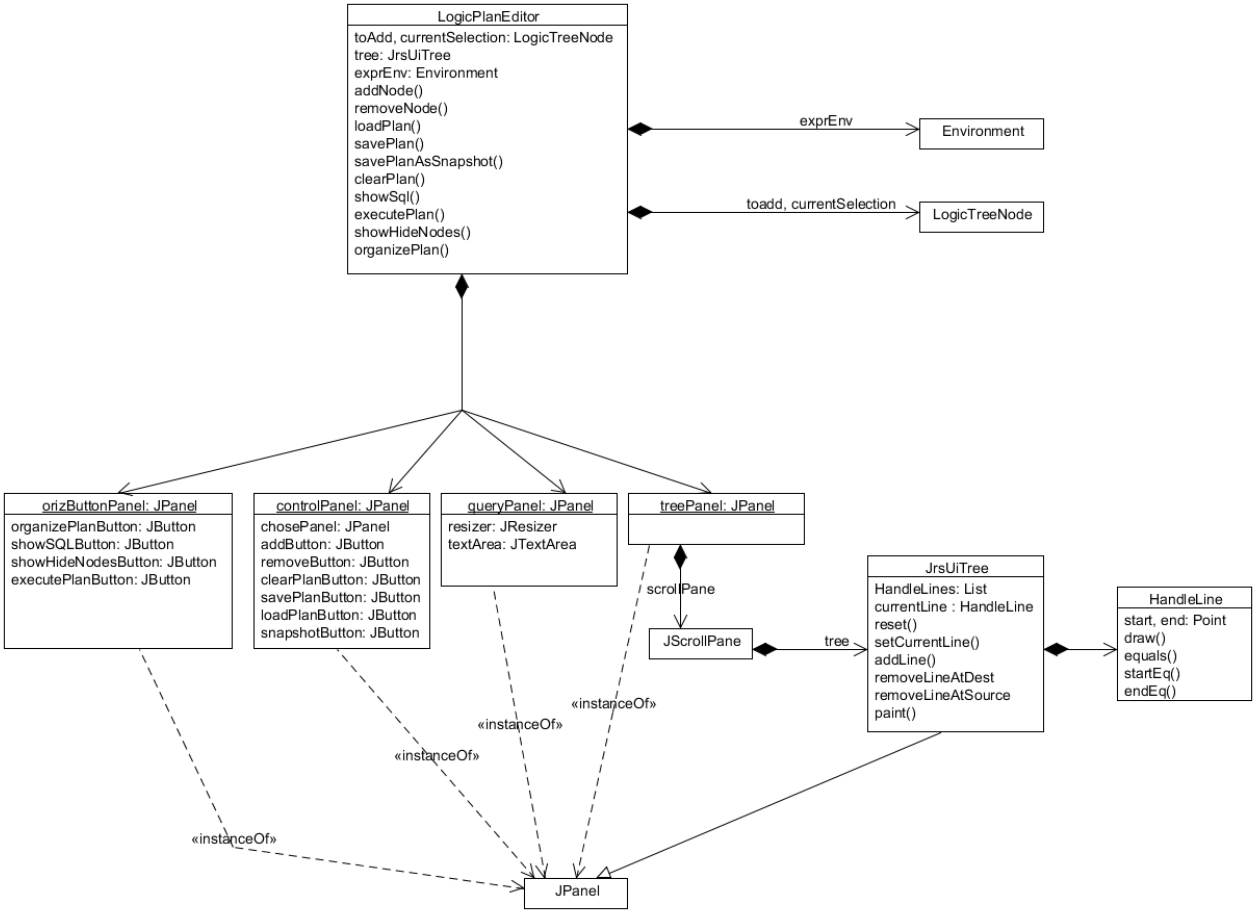

Figura 5.1 Diagramma delle classi dell'editor di piani logici

Dal diagramma si vede chiaramente come la divisione in aree data nel capitolo 3 abbia una diretta corrispondenza con le istanze di **JPanel** che vanno a comporre la finestra di dialogo. **JPanel** è la classe di *Swing* per i pannelli. Segue una breve descrizione dei singoli pannelli.

- **orizButtonPanel** è un'area in cui sono contenuti su una riga i bottoni posti tra **queryPanel** e **treePanel**, cioè i bottoni per le funzionalità **organizePlan**, **showSQL**, **showHideNodes**, **executePlan**.
- **controlPanel** contiene il bottone per aggiungere un nodo **addButton**, un altro pannello **chosePanel**, indicato nell'interfaccia come *Select Operator*, che contiene per ogni tipo di nodo un **RadioButton**, e i bottoni relativi alle funzionalità **addNode**, **removeNode**, **clearPlan**, **savePlan**, **loadPlan**, **snapshot**.
- **queryPanel** contiene l'area per la visualizzazione dei risultati, indicata con **Query** o **Output**, realizzata come un'istanza di **JTextArea**. Inoltre contiene una barra istanza di **JResizer** che permette di ridimensionare l'area.
- **treePanel** è l'area che contiene l'albero, e quindi i nodi e gli archi. I nodi sono istanze di **LogicTreeNode**, mentre per la gestione degli archi ètata definita una classe **JrsUiTree**, sottoclasse di **Jpanel**. Allo scopo di aggiungere al pannello le barre di scorrimento, si è fatto uso della classe **JScrollPane**.

Il trattamento degli archi è realizzato nella classe **JrsUiTree**, mentre i nodi possono essere normalmente aggiunti a un **JPanel** in quanto sottoclassi di **JToggleButton**. Un singolo arco è un'istanza della classe **HandleLine**. La descrizione di **JrsUiTree** è la seguente :

- un campo **HandleLines**, lista di istanze di **HandleLine**, e quindi di archi.
- **currentLine** un'istanza di **HandleLine**, cioè un arco, in particolare l'arco che si sta attualmente disegnando.
- un metodo **reset()** per cancellare tutti gli archi.
- un metodo **setCurrentLine()** per aggiornare l'arco che si sta correntemente disegnando.
- un metodo **addLine()** per aggiungere un arco.
- due metodi **removeLineAtDest**, **removeLineAtSource** per la rimozione di un singolo arco identificato dalle coordinate di arrivo o da quelle di partenza rispettivamente.
- un metodo **paint()**, che sovrascrive il metodo di **JPanel**, responsabile di disegnare tutti gli archi come linee.

La classe per gestire un singolo arco è **HandleLine**, descritta come segue :

- due campi **start**, **end**, istanze di **Point**, per le coordinate di partenza e di arrivo dell'arco.
- un metodo **draw()** per disegnare il singolo arco. Questo metodo viene utilizzato nel metodo **paint()** di **JrsUiTree** e disegna una linea da **start** a **end**.
- tre metodi per il confonto di archi **equals()**, **startEq()**, **endEq()**, per confrontare due archi o deteminare se un arco ha determinate coordinate di partenza o di arrivo rispettivamente.

La rappresentazione grafica dell'albero viene quindi realizzata nel metodo **paint()** di **JrsUiTree**. Il semplice algoritmo di tale metodo è definito in Figura 5.2.

```
super()
for each HandleLine h in HandleLines do {
  h.draw()
}
```

Figura 5.2 Algoritmo per disegnare l'albero

Tutte le funzionalità nominate in occasione della descrizione dei vari pannelli sono realizzate nella classe **LogicPlanEditor**, sottoclasse di **JDialog**, la classe di *Swing* per le finestre di dialogo. Segue una descrizione della classe:

- due campi **toAdd**, **currentSelection**, riferimenti a **LogicTreeNode**, che indicano il nodo da aggiungere e il nodo correntemente selezionato.
- un campo **tree**, istanza di JrsUiTree, descritto in precedenza.
- un campo **exprEnv**, istanza di **Environment**, che gestisce l'ambiente delle espressioni definite nel piano.
- due metodi **addNode()** e **removeNode()** per aggiungere o rimuovere un nodo.
- due metodi **loadPlan()** e **savePlan()** per caricare o salvare un piano su disco. Tali metodi fanno uso del **JFileChooser** per far indicare all'utente il nome del file da caricare o salvare. L'operazione poi viene effettuata utilizzando le funzionalità del **Serializer** di Java.
- un metodo **savePlanAsSnapshot()** per salvare un'immagine in formato *jpeg* del piano.

- un metodo **clearPlan()** per cancellare il piano e cioè rimuovere tutti i nodi e tutti gli archi.
- un metodo **showSql()** che converte il piano logico in **SQL**. Una descrizione dettagliata di tale metodo è fornita in seguito.
- un metodo **executePlan()** per eseguire un piano logico. Questo metodo converte prima il piano in **SQL** utlizzando il metodo definito in predenza, e quindi fa eseguire la query ottenuta al JRS.
- un metodo **showHideNodes()** per nascondere o visualizzare i controlli relativi ai nodi.
- un metodo **organizePlan()** per ordinare i nodi e gli archi di un piano, che verrà descritto in maggior dettaglio in seguito.

Due metodi in particolare meritano una descrizione più dettagliata, **showSql** che converte un piano logico in **SQL** e **organizePlan** che riorganizza la disposizione dei nodi di un piano.

Conversione di un piano logico in SQL

Nel JRS non è prevista la possibilità di eseguire un piano logico, ma essendo possibile eseguire un'interrogazione formulata in **SQL**, si può aggirare tale problema generando un'interrogazione **SQL** equivalente al piano. Dal momento che le interrogazioni formulabili in algebra relazionale sono un sottoinsieme di quelle formulabili in SQL, tale traduzione è sempre possibile. Un primo semplice algoritmo a cui si può pensare consiste nel convertire ogni singolo nodo in un'interrogazione, e quindi annidare opportunamente i risultati in varie *sottoselect*. Tale soluzione però oltre a essere banale, conduce a interrogazioni poco leggibili per l'utente, e non è in linea con lo scopo didattico del progetto. Si è quindi utilizzato un algoritmo più complesso, che consente di convertire un piano in un'interrogazione **SQL** concisa e leggibile. In quest'algoritmo, non si convertono singoli nodi ma sottoalberi, i quali vengono individuati da un apposito algoritmo. In presenza di più sottoalberi, tutti i sottoalberi con radice diversa da quella dell'albero originario saranno inclusi in viste, alle quali poi faranno riferimento le interrogazioni risultanti dalla conversione dei sottoal-

beri successivi. L'algoritmo per la conversione fa quindi uso di due ulteriori algoritmi, uno per la determinazione dei sottoalberi da tradurre, e un altro per la traduzione di tali sottoalberi. Nella Figura 5.3 è possibile vedere l'algoritmo in pseudo codice.

```
1: LogicTreeNode cloned = root.clone()
2: String views = generateViews(cloned)
3: if not queryComplete {
4:    query=subTreeToSQL(cloned)
5: }
6: query = refactorSql(views+query)
7: textArea.print(query)
```

Figura 5.3 Algoritmo per la conversione in SQL

Le parti più interessanti dell'algoritmo sono i due algoritmi menzionati. Il metodo **generateViews()** realizza l'algoritmo per determinare quali sottoalberi convertire, e in che ordine, ed eventualmente utilizzare le viste. Nella Figura 5.4 è possibile vedere l'algoritmo in pseudo codice.

```
1 : List nodes = reverse(visitBFS(root))
2 : for each node in nodes do {
3 :   if node instanceof SetOperators OR node instanceof GroupBy {
4 :       if node instanceof GroupBy
5 :        find an ancestor anc of node such that
6 :          all ancestors of node and descendants of anc
7 :            instanceof (UnaryOperators except GroupBy)
8 :       viewName = "LTV"+sequentialNumber()
9 :       view = subTreeToSQL(anc)
10:       make anc a leaf
11:       for each ancestor of anc do {
12:          add reference to viewName
13:       }
14:    }
15:    if not anc = root {
16:       make the view viewName
17:       add the view to result
18:    }
19:    else {
20:      queryComplete=true
21:      add the query in view to result
22:    }
23: }
```

Figura 5.4 Algoritmo per generare le viste

Segue un breve commento per l'algoritmo mostrato:

– nella riga 1, si esegue una visita BFS rovesciata dell'albero. In questo modo la ricerca dei sottoalberi rispetta le dipendenze tra nodi, infatti ogni nodo

dipende dai suoi operandi che nell'albero sono i nodi figli e quindi posti a un livello di profondità in più. Con una visita BFS rovsesciata si procede in ordine di profondità, dal nodo più profondo al nodo radice. Per quanto detto in precedenza, ciò implica che quando nella lista risultato della visita si giunge a un nodo, tutti i suoi figli sono già stati visitati.

- nella riga 3 osserviamo le condizioni che rendono necessaria la creazione di una vista, e cioè la presenza di operatori insiemistici o dell'operatore **GroupBy**.
- nelle righe 4-7, nel caso in cui la vista sia per un nodo del tipo **GroupBy**, si tenta di risalire nell'albero il più possibile, allo scopo di ridurre il numero di viste. Lungo il percorso sono ammessi solo nodi di operatori unari diversi da **GroupBy**.
- nelle righe 8-9 si genera l'interrogazione relativa alla vista utilizzando il metodo **subTreeToSQL**.
- nelle righe 10-13 si trasforma l'albero logico, in modo che le successive traduzioni tengano conto della vista appena generata. Il nodo radice della vista diviene una foglia nell'albero, che rappresenta proprio la vista generata. In seguito si aggiornano tutti gli antenati di tale nodo in modo da contenere il riferimento alla vista. Quello che si fa in pratica, è di aggiungere il nome della vista come prefisso a tutti gli attributi della vista presenti negli antenati.
- nelle righe 15-22 si aggiunge la vista al risultato, e se si è giunti al nodo radice, si aggiunge l'interrogazione senza trasformarla in vista.

Al fine di generare le viste si applicano quindi delle trasformazioni all'albero logico da convertire, per cui i nodi radice delle viste diventano delle foglie. Per non modificare l'albero logico iniziale, si copia tale albero in un clone e si opera sul clone. Per esempio se si considera l'albero in Figura 5.5, si può osservare come viene trasformato al termine della generazione delle viste. L'interrogazione generata dall'editor per quest'esempio infatti è:

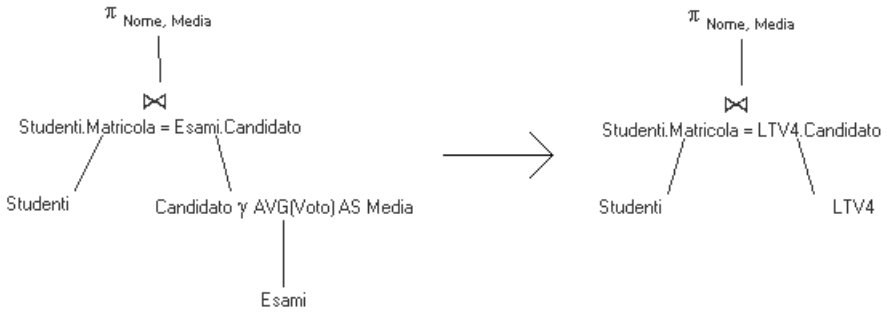

Figura 5.5 Trasformazione dell'albero nella generazione delle viste

```
CREATE VIEW LTV4 AS
SELECT      Candidato, AVG(Voto) AS Media
FROM        Esami
GROUP BY    Candidato
;
SELECT DISTINCT Nome, Media
FROM        Studenti, LTV4
WHERE       Matricola = Candidato
;
```

Ora che i sottoalberi sono individuati, è possibile convertirli tutti in **SQL**, e ogni sottoalbero è traducibile in un'interrogazione con le sole clausole **Select**, **From**, **Where**, **Group By**, **Having**, **Order By** senza necessità di sottoselect o viste. Nella Figura 5.6 è rappresentato l'algoritmo in pseudo codice.

Segue una descrizione dell'algoritmo mostrato:

- nelle righe 1-18 vengono generate le clausole **Select** e **From**, con gli attributi di **Select** presi dal nodo radice e quelli di **From** dai nomi di tabella e variabili di correlazione delle foglie.
- nelle righe 19-27 si generano le clausole **Where** e **Having**, unendo in *AND* tutte le condizioni ricavate dai nodi di tipo restrizione o giunzione.
- nelle righe 28-29 si genera la clausola **Order By** se è presente un nodo di tipo **Order By**. Gli attributi di ordinamento sono ricavati dai parametri del nodo.
- nelle righe 30-34 si determina se bisogna aggiungere **Distinct** all'interrogazione, controllando se è presente una proiezione (qui chiamata distinct) o un

```
1 : List selectAttributes = root.getAttributes()
2 : String select="SELECT"
3 : if selectAttributes ≡ union of all leaf attributes
4 :    select+=" *"
5 : else for each attribute in selectAttributes do {
6 :    if attribute is an expression
7 :       attribute=instantiateExpressions(attribute)
8 :    if attribute is renamed
9 :       select+=getAlias(attribute)+" AS "+attribute+", "
10:    else select+=attribute+", "
11: }
12: from="FROM"
13: List leafs = getLeafs(root)
14: for each leaf in leafs do {
15:    if leaf has correlation
16:       from+=leaf.getTableName()+" "+leaf.getCorrelation()
17:    else from+=leaf.getTableName()
18: }
19: where="WHERE"
20: having="HAVING"
21: List restrictions = getAllRestrictions(root)
22: List joins = getAllJoins(root)
23: for each node in restrictions U joins do {
24:    if condition has aggregateFunctions
25:       having+=node.condition+" AND ";
26:    else where+=node.condition+" AND ";
27: }
28: if root instanceof orderBy
29:    orderby="ORDER BY" + all root parameters
30: List nodes = visit(root)
31: if distinct ∈  nodes OR (groupBy ∈  nodes AND
32:    selectAttributes contains groupBy.attributes)
33:    distinct="DISTINCT"
34: else distinct=""
35: query=select+distinct+from+where+having+orderby
36: query=refactorSQL(query, root)
```

Figura 5.6 Algoritmo per generare l'SQL relativo a un sottoalbero

groupby e non tutti gli attributi di raggruppamento sono attributi di **Select**.

– nelle righe 35-36 si compone la query e quindi si esegue il metodo **refactorSQL** che ha il compito di eliminare i prefissi inutili e di riscrivere l'interrogazione in modo leggibile.

Il metodo organizePlan

Per riorganizzare il posizionamento della rappresentazione grafica dei nodi di un albero, che possono anche avere dimensioni differenti sia in larghezza che altezza, è stato definito un algoritmo il cui pseudo-codice è visibile in Figura 5.7. Nel JRS è presente un algoritmo simile, usato per visualizzare i piani di accesso, ma su nodi più semplici, di dimensione fissa e che hanno come etichetta il solo nome dell'operatore. Il modo di procedere dell'algoritmo realizzato per l'editor è di tipo iterativo: dopo aver eseguito una visita per individuare le foglie, si calcolano le coordinate degli estremi e le dimensioni dell'albero, facilmente ricavabili dalla disposizione dei nodi foglia.

– nelle righe 1-3 sono dichiarate le variabili e impostati i margini, e si trovano tutte le foglie dell'albero.
– nelle righe 3-6 si posizionano le foglie, calcolando le coordinate come indicato.
– nella riga 7 si aggiunge un nodo non foglia, padre del nodo foglia che su cui si sta correntemente iterando, alla lista nodes.
– nelle righe 9-17 si posizionano i nodi non foglia, e se hanno due figli sono posizionati al centro sopra i due figli, altrimenti se hanno un figlio solo, vengono posizionati direttamente sopra al nodo figlio. I nodi vengono posizionati usando le coordinate del centro del nodo e non dell'angolo in alto a sinistra.
– nelle righe 18-27 vedo se i nodi figli sinistri si sovrappongono con i nodi fratelli, e in tal caso li sposto entrambi.
– nelle righe 29-36 vedo se dopo aver spostato i nodi come sopra, ci sono ancora nodi sovrapposti. La visita usata è una visita DFS da sinistra a

```
1 : List nodes
2 : List leafs = getLeafs(root)
3 : int x = 140, y = 10
3 : for each leaf in leafs do {
4 :   place the node's center at coordinates:
5 :     x, y+level*height*2+10
6 :   x+=width+10
7 :   add leaf.parent to nodes
8 : }
9 : while not nodes is empty do {
10:   if node has 2 childs place node center at:
11:     (node.left.x+node.right.x)/2,
12:       y+node.level*height*2+10
13:   if node has 1 child place node center at:
14:     child.x,
15:     y+node.level*height*2+10
16:   add node.parent to nodes
17:   remove node from nodes
17: }
18: nodes = visit(root)
19: for each node in nodes do {
20:   if node is left of node.parent
21:      brother=parent.right
22:   if brother != null {
23:     if node overlaps with brother {
24:       move subtree of node overlap/2 to the left
25:       move subtree of brother overlap/2 to the right
26:     }
27:   }
28: }
29: for each node in nodes do{
30:   for each nodeIntern in nodes do {
31:     if node.level = nodeIntern.level and
32:        node overlaps with nodeIntern {
33:       move subtree of nodeIntern overlap to the right
34:     }
35:   }
36: }
37: repaint tree
```

Figura 5.7 Algoritmo per riordinare i nodi di un albero

destra, quindi spostando ora i nodi che si sovrappongono verso destra, non ci sono più nodi sovrapposti e l'albero è anche graficamente bilanciato.

Gli alberi così riordinati sono ora ben leggibili e ordinati, senza nodi sovrapposti e con archi verticali in caso di figli unici, e in caso di due figli, il padre è posto un livello sopra al centro tra i due figli. Tutti gli alberi mostrati nelle figure della tesi sono stati ordinati con questo algoritmo.

5.2.2 Diagramma UML delle classi relative agli alberi logici

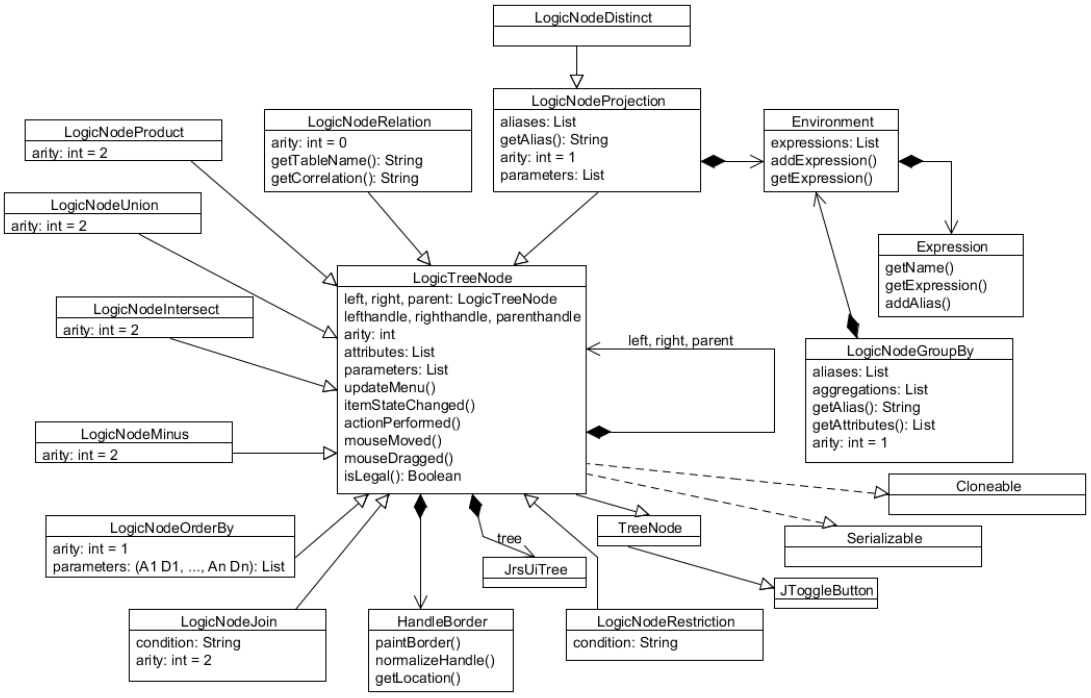

Figura 5.8 Diagramma delle classi del nodo logico

Nel diagramma è evidenziato il fatto che a ogni tipo di operatore algebrico, corrisponde un particolare tipo di nodo dell'albero logico, al quale corrisponde una classe, realizzata come sottoclasse di **LogicTreeNode**. Si procede con una descrizione ordinata a partire dalle sottoclassi per finire con la descrizione della classe **LogicTreeNode**. Per tutte le classi è definito il campo **arity** che determina l'arietà del nodo, ed è ovviamente determinata dall'arietà dell'operatore rappresentato. Inoltre è importante sottolineare come nel metodo **updateMenu**, oltre

a impostare il menu contestuale del nodo, vengono calcolati gli attributi del nodo, tenendo presente gli attributi dei figli. Grazie al calcolo del tipo di ogni operatore rappresentato dal nodo, è stato possibile realizzare menu contestuali che guidano l'utente verso scelte corrette. Gli attributi che determinano tale tipo sono contenuti in una lista **attributes**, mentre i parametri dell'operatore sono contenuti in un'altra lista **parameters**. Ci sono alcune classi che meritano una breve descrizione dettagliata:

- **LogicNodeOrderBy** eredita la lista degli attributi dal figlio, e gli attributi di ordinamento sono elencati nella lista **parameters** secondo la forma *Attributo*, *ASC* o *DESC* a seconda che l'ordine sia ascendente o discendente.
- **LogicNodeJoin** e **LogicNodeRestriction** hanno un campo **condition** che contiene la condizione di restrizione o giunzione. Come si è visto, tale campo risulta importante per la generazione dell'interrogazione in **SQL**.
- **LogicNodeRelation** ha due metodi **getTableName** e **getCorrelation** per accedere al nome della tabella e alla variabile di correlazione.
- **LogicNodeProjection** ha un campo **aliases** di tipo lista che contiene la lista degli alias, utilizzati per memorizzare le rinominazioni. Un alias è una stringa che contiene 3 *token* separati da spazio:
  - un primo token per il vecchio nome dell'attributo.
  - un secondo token per il nuovo nome dell'attributo.
  - un ultimo token per il prefisso dell'attributo, memorizzato per comodità.
- **LogicNodeGroupBy** oltre al campo **aliases**, contiene un campo **aggregations** che memorizza le funzioni di aggregazione.

La classe LogicTreeNode

LogicTreeNode è la superclasse per tutti i nodi di alberi logici, ed è strutturata come segue:

- tre campi **left**, **right**, **parent**, riferimenti a **LogicTreeNode**, per indicare i nodi figlio sinitro, figlio destro e padre rispettivamente.

- tre campi **lefthandle**, **righthandle**, **parenthandle** che indicano gli *handle* di arrivo dell'arco proveniente dal nodo padre e di partenza per gli archi diretti verso i nodi figli, e con appositi metodi è possibile calcolarne le coordinate.
- un campo **arity** che viene impostato nelle sottoclassi e che indica l'arietà dell'operatore algebrico associato al nodo.
- un campo **attributes** che contiene la lista dei nomi degli attributi, sempre scritti con prefisso il nome della tabella. La scelta di prefissare sempre gli attributi è dovuta al fatto che è molto più semplice eliminare un prefisso piuttosto che doverselo andare a cercare visitando l'albero. Tali prefissi tornano utili quando si vuole disambiguare tra due attributi di uguale nome, oltre che per esempio nel calcolo della condizione di giunzione in caso di equi join.
- un campo **parameters** che contiene i *parametri* relativi al nodo, in un formato diverso a seconda del tipo di nodo.
- un metodo **updateMenu** utilizzato per creare il *menu contestuale* relativo a un nodo. Questo metodo inizializza anche il valore della lista **attributes**, e dal momento che viene chiamato in seguito all'aggiunta di un arco, compie anche le verifiche sintattiche sugli operandi. Per esempio negli operatori insiemistici, pur non essendoci un menu, questo metodo è responsabile di verificare che i figli abbiano lo stesso tipo, e in caso contrario di stampare il messaggio di errore.
- due metodi **actionPerformed** e **itemStateChanged** per gestire gli *eventi* dei *menu contestuali*. Essendo un'applicazione giudata dagli *eventi*, saranno questi metodi a aggiornare i parametri del nodo e eventualmente il *menu contestuale*.
- due metodi **mouseMoved** e **mouseDragged** per gestire gli eventi derivanti da azioni del mouse, al fine di spostare i nodi o di aggiungere archi.
- un metodo **isLegal** che verifica se l'arco che si sta aggiungendo è conforme alle regole sinstattiche del nodo e alle regole per gli aberi binari. Verifica tra l'altro che l'arco non formi un ciclo, che non sia un secondo arco entrante, e in questo metodo si impone anche la condizione sul nodo τ perchè sia nodo

radice, e il nodo π^b può essere solo radice o figlio del nodo τ.

Tra le interfacce implementate, nel diagramma sono evidenziate le classi:

- **Cloneable** per rendere possibile la clonazione dell'albero in occasione della traduzione in **SQL**.
- **Serializable** per rendere possibile la conversione dei singoli nodi e quindi di tutto un albero in file da scrivere su disco e viceversa.

Nel diagramma sono presenti inoltre i riferimenti a altre due classi:

La classe HandleBorder

Questa classe serve per realizzare il meccanismo degli *handle* per l'aggiunta o la rimozione degli archi. Gli handle sono localizzati sul *bordo* del nodo, disegnati come piccoli rettangoli. Sono evidenziati tre metodi:

- **paintBorder** disegna il bordo, e se il puntatore del mouse è all'interno del nodo, disegna il bordo con i rettangoli degli *handle*.
- **normalizeHandle** calcola le coordinate da dare a un estremo di un arco che deve essere piazzato all'interno di un determinato *handle*.
- **getLocation** compie l'operazione inversa, cioè date delle coordinate determina in quale *handle* sono comprese.

La classe Environment

Nei nodi di tipo **LogicNodeProjection** e **LogicNodeGroupBy** è possibile definire delle espressioni. La classe **Environment** permette di memorizzare e gestire queste espressioni, al fine di poter risalire alla loro definizione dato il nome. È molto utile nella generazione del **SQL**. Nel diagramma si evidenzia:

- un campo **expressions** di tipo lista, che contiene le espressioni di questo ambiente.
- un metodo **addExpression** per aggiungere un'espressione all'ambiente.
- un metodo **getExpression** per trovare la definizione di un'espressione dato il nome.

Una singola espressione è rappresentata da un'istanza della classe **Expression**.

5.2.3 Diagramma UML delle classi relative agli alberi fisici

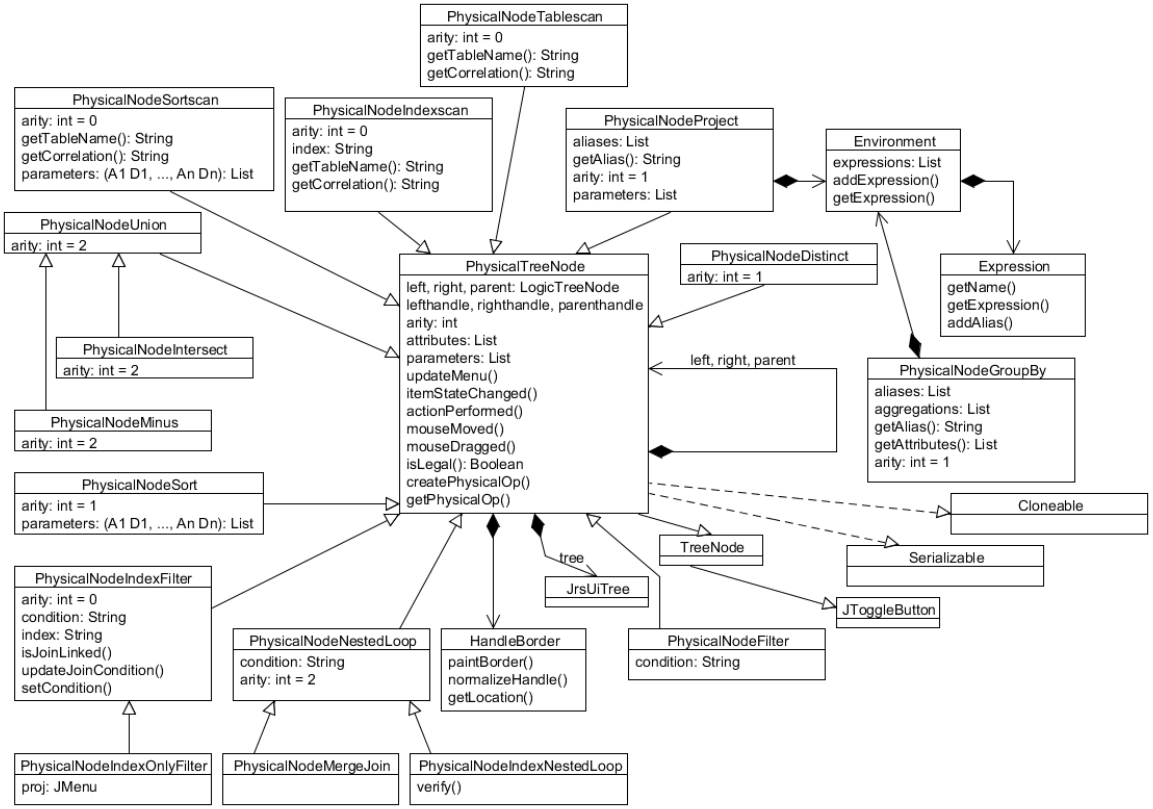

Figura 5.9 Diagramma delle classi del nodo fisico

Per una descrizione di metodi e campi uguali al caso degli alberi logici, si rimanda alla sezione precedente. Vi sono però alcune differenze, oltre che sulla struttura dei menu, caratterizzate dalla presenza dei metodi **createPhysicalOp** e **getPhysicalOp**.

– **createPhysicalOp** crea l'operatore fisico del JRS rappresentato da questo nodo, con i parametri corretti. Tale operatore viene utilizzato nel metodo **executePlan** dell'editor di alberi fisici, per creare un albero di iteratori al fine di eseguire un albero fisico.
– **getPhysicalOp** nella fase di creazione dell'albero degli iteratori, con questo metodo si ottiene l'operatore fisico associato.

Il semplice algoritmo per eseguire un albero fisico è rappresentato in Figura 5.10. Si fornisce una breve descrizione:

```
1 : List nodes = reverse(visitBFS(root))
2 : for each node in nodes do {
3 :    node.createPhysicalOp()
4 : }
5 : rootOp = root.getPhysicalOp()
6 : set header using rootOp.type
7 : while not rootOp.isDone() do {
8 :    rec = rootOp.next()
9 :    output+=rec
10: }
11: display(output)
12: textArea.append(output)
```

Figura 5.10 Algoritmo per eseguire un albero fisico

– nella riga 1, si procede a una visita BFS rovesciata. In questo modo gli operatori vengono creati rispettando le dipendenze tra i nodi, e quindi tra gli operatori fisici.

– nelle righe 2-4 si creano tutti gli operatori fisici associati ai nodi.

– nella riga 5 si memorizza l'operatore relativo al nodo radice.

– nella riga 6 si calcola l'intestazione per il risultato a partire dal tipo di operatore del nodo radice.

– nelle righe 7-10 si procede all'iterazione, seguendo l'interfaccia di iteratore.

– nelle righe 11-12 si stampa il risultato sia nell'area di Output dell'editor che in una nuova finestra.

Dovendo eseguire alberi fisici, si è proceduti anche a una revisione degli operatori fisici del JRS, realizzati per essere utilizzati dall'ottimizzatore, e quindi non sempre correttamente funzionanti in caso di utilizzi differenti. Inoltre nei vari test sono stati individuati anche alcuni bug del JRS, che di conseguenza risulta più stabile nel suo complesso.

5.3 Conclusioni

In questo capitolo è stata fornita una visione d'insieme sulla realizzazione dell'editore, mostrando nel dettaglio lo schema delle classi e gli algoritmi relativi ad alcune funzionalità ritenute più significative, quali la conversione di un albero logico in **SQL** o l'esecuzione di un albero fisico. Nel prossimo capitolo si mostra l'estendibilità del sistema, fornendo anche un esempio di estensione.

Capitolo 6

MODIFICABILITÀ ED ESTENDIBILITÀ DEL SISTEMA

Questo capitolo vuole non solo dimostrare che il sistema è modificabile e estendibile, ma vuole anche fornire una breve guida per compiere tali operazioni. Nella prima parte si descrive come modificare alcune operazioni che probabilmente richiederanno di essere continuamente aggiornate, per poi mostrare l'estendibilità con un esempio.

6.1 Modificabilità del sistema

Nel precedente capitolo si è già data una descrizione dei vari metodi presenti nell'editor, ora per alcuni si da una descrizione un po' più dettagliata mirata a fornire utili indicazioni per una eventuale modifica degli stessi.

6.1.1 Le etichette dei nodi

Il testo scritto sulle etichette dei nodi è formattato in **HTML**. Dato un particolare nodo, e cioè la sottoclasse che corrisponde all'operatore, vi sono in generale due metodi responsabili di generare il testo dell'etichetta del nodo:

– **typeToString** fornisce la prima parte dell'etichetta, che comprende il simbolo dell'operatore, non dipendente dai parametri, eccezion fatta per i nodi foglia dove questo metodo è responsabile per la generazione dell'etichetta fino ai nomi di tabella e alle variabili di correlazione.
– **generateTexts** genera il testo della seconda parte dell'etichetta, in generale è la parte dipendente dai parametri.

Quindi a seconda di quale parte dell'etichetta si vuole modificare, occorre modificare l'uno o l'altro metodo.

6.1.2 I menu contestuali

I menu contestuali sono impostati inizialmente nel metodo **updateMenu** ma in seguito possono essere aggiornati per adattarli ai parametri immessi tramite i menu stessi. I metodi coinvolti sono quindi:

– **updateMenu** per generare l'intero menu non appena il nodo ha un numero di archi pari alla sua arietà, o per reimpostare il menu nel caso un discendente sia stato modificato.
– **itemStateChanged** è il gestore degli eventi per i controlli nei menu che possono essere selezionati e deselezionati. Tali eventi corripondono all'immissione dei parametri, e possono determinare un aggiornamento del menu. Si veda per esempio nel nodo π il sottomenu **AS**, aggiornato ogni volta che si aggiunge o rimuove un attributo dai parametri.
– **actionPerformed** in modo analogo al precedente ma per menu senza possibilità di impostare una selezione, come per esempio le scelte degli indici.

6.1.3 Informazioni stampate in Node Info

Le informazioni della finestra **Node Info**, per entrambi gli editor, vengono generate nelle rispettive superclassi dei nodi logici e fisici. Il metodo è **mouse_Clicked**, il gestore di eventi per i clic del mouse. All'interno di tale metodo si osserva come le informazioni vengano generate in ordine e poi aggiunte a una stringa, che sarà poi stampata dalla classe **NodeInfo**.

6.1.4 Campi dei nodi passati al nodo padre

Nel caso in cui si voglia aggiungere un campo da passare al nodo padre, si consiglia di agire come segue:

- aggiungere il campo nella superclasse.
- nel metodo **updateMenu** si può calcolare il nuovo valore del campo a partire dai dati dei nodi figli.
- nel costruttore del nodo, si procede a inizializzare il campo a un valore nullo o vuoto.

Per esempio, tale procedura è da seguire se si vuole aggiungere ai nodi fisici la stima dei costi.

6.1.5 Controlli sull'aggiunta di archi

La verfica sull'ammissibilità di un arco che si sta aggiungendo avviene in un solo metodo, definito nelle superclassi dei nodi logici e fisici, **isLegal**. Per aggiungere quindi altre condizioni, o modificare quelle presenti, è sufficiente modificare questo metodo. **isLegal** restituisce **false** se il nodo non può essere aggiunto, **true** altrimenti.

6.1.6 La classe Constants

Nella classe **Constants** sono contenute varie costanti, che all'occorenza possono essere modificate. Tra le costanti sono presenti la lista dei token, il numero di spazi per allineare una query generata, e così via.

6.1.7 I messaggi di errore

Tali messaggi sono sempre visualizzati utilizzando la classe **JOptionPane**. Per determinare dove un messaggio viene stampato, è sufficiente controllare nel gestore dell'evento che lo ha generato.

6.2 Estendibilità del sistema

Si mostra come estendere il sistema con laggiunta di un operatore logico, che permette di eseguire la divisione tra due relazioni R(A, B) e S(B) di cui una con 2 attributi e l'altra con solo il secondo degli attributi della prima. R ÷ S ritorna una relazione con attributo A e elementi i valori di A che sono presenti in R con tutti i valori B in S. Si riporta la definizione generale:

$R \div S = \{w \mid \forall s \in S.w \circ s \in R\}$

È un operatore derivato, e si può derivare dagli operatori primitivi come segue:

$\pi_A(R) - R_1$ con $R_1 = \pi_A((\pi_A(R) \times S) - R)$.

con (A, B) gli attributi di R e B attributo di S. Il risultato è una relazione $W = R \div S$ con attributi A.

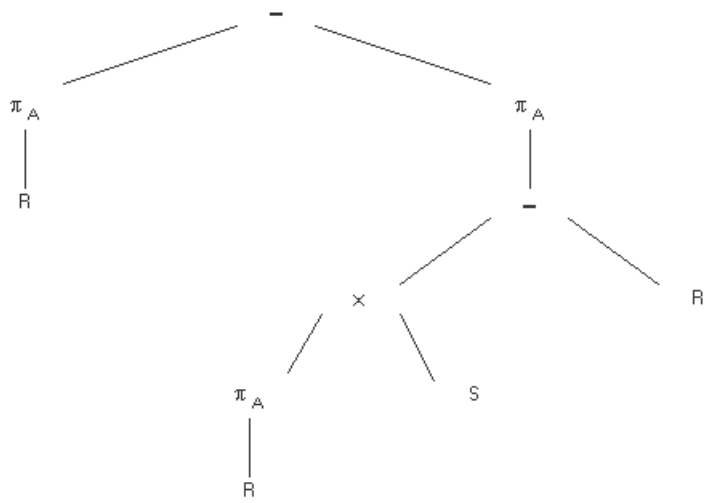

Figura 6.1 Piano logico che deriva l'operatore di divisione

Un esempio dell'applicazione di tale definizione viene dato nella Figura 6.1. Si noti come tale piano utilizzi diversi nodi per realizzare un singolo operatore. Risulta quindi evidente il vantaggio di estendere il sistema con l'operatore in questione.

6.2.1 Aggiunta della classe relativa al nuovo nodo

Avendo semplificato l'operatore, anche la classe relativa al nuovo nodo risulterà semplice. Prima di commentare il codice, alcune considerazioni:

- dal momento che si vuole aggiungere un nuovo nodo logico, si crea un nuovo file dal nome **LogicNodeDivision**, poichè tutti i nomi delle classi dei nodi logici iniziano con **LogicNode**.
- la classe deve estendere **LogicTreeNode**.
- il costruttore, oltre alla chiamata al costruttore della superclasse, deve almeno impostare l'arietà del nodo, in questo caso a 2.

Listing 6.1 Codice Java per il nodo logico divisione

```
1  package jrsui;
2
3  import java.awt.Font;
4  import java.util.LinkedList;
5  import javax.swing.*;
6
7  public class LogicNodeDivision extends LogicTreeNode {
8
9    private static final long serialVersionUID=20241702641564231 50L;
10
11   public LogicNodeDivision(Font treeFont,LogicPlanEditor _dialog){
12     super(treeFont, _dialog);
13     arity = 2;
14   }
15
16   public String typeToString() {
17     return "<html><font size=+0>\u00F7</font>";
18   }
19
20   public String typeToStringSimple() {
21     return "\u00F7 ";
22   }
23
24   public void updateMenu(LogicTreeNode child) {
25     super.updateMenu(child);
```

```
26      if (child == null)
27        return;
28      if (left == null || right == null) {
29        attributes = new LinkedList<String>();
30        setText(typeToString());
31        setToolTipText(typeToString());
32        return;
33      } else {// ha 2 figli
34  // si verifica che gli attributi siano
35  // esattamente come richiesti per questo
36  // operatore : sinistro con due attributi e
37  // destro con il secondo di tali attributi
38        if (left.getAttributes().size() == 2
39          && right.getAttributes().size() == 1)
40          if (Utility.suffix(left.getAttributes().get(1)).equals(
41              Utility.suffix((right.getAttributes().get(0))))) {
42                attributes = new LinkedList<String>();
43                attributes.add(left.getAttributes().get(0));
44                return;
45            }
46          // ha due figli ma non rispettano le condizioni
47        attributes = new LinkedList<String>();
48        JOptionPane
49          .showMessageDialog(
50          this,
51          "Error : The left operand ...",
52          "Division Operator", JOptionPane.ERROR_MESSAGE);
53          return;
54      }
55    }
56  }
```

Segue un commento delle parti più significative del codice:

- nella riga 7 si definisce un numero seriale che è richiesto dal **Serializer** per poter scrivere oggetti di questa classe su file.
- nelle righe 16-22 si definisce il metodo che associa l'etichetta al nodo, in questo caso è un semplice simbolo di divisione ÷. Il codice **u00F7** è il *codice unicode* associato al carattere per stampare ÷.

– nelle righe 24-33 si definisce la prima parte del metodo **updateMenu**, che controlla se il nodo ha 2 figli.

– nelle righe 38-41, stabilito che ci sono 2 figli, vi sono i controlli per verificare che gli attributi dei nodi figli corrispondano alla forma richiesta.

– nelle righe 42-43, avendo verificato tutte le condizioni di correttezza, si imposta il tipo del risultato del nodo nella lista **attributes**, e cioè il primo attributo dell'operando sinistro.

– nelle righe 47-52 si da un messaggio di errore nel caso in cui il nodo abbia 2 figli che però non corrispondono alla forma richiesta.

6.2.2 L'aggiunta del nodo al menu dell'editor

Ora che la classe relativa al nodo è stata creata, bisogna aggiungere il pulsante all'editor per poter aggiungere questo tipo di nodo a un piano logico. Si procede quindi a modificare la classe **LogicPlanEditor**. Nel metodo **jbInit** si aggiunge il pulsante al riquadro **Select Operator**.

Listing 6.2 Codice Java per l'aggiunta del pulsante per il nodo divisione

```
1 JRadioButtonMenuItem divisionButton2 = new JRadioButtonMenuItem();
2
3 chosePanel.add(divisionButton2);
4 divisionButton2.addItemListener(new DivisionButtonItemListener());
5 RadioButtonGroup.add(divisionButton2);
6 divisionButton2.setFont(jrsTimesFont.deriveFont(Font.PLAIN, 15));
7 divisionButton2.setText(" \u00F7 ");
```

Quindi si definisce il gestore per l'evento del bottone appena aggiunto.

Listing 6.3 Codice Java per gestire l'evento del pulsante

```
1 protected void divisionButton_itemStateChanged(ItemEvent e) {
2   if(e.getStateChange() == ItemEvent.SELECTED) {
3   addButton.setFont(jrsButtonFont.deriveFont(Font.PLAIN, 12));
4   addButton.setText("<html><b>Operator </b><br><font size=+2>...")
5   addWhich = NodeType.DIVISION;
6   addNode();
7   }
8 }
```

Restano altre due piccole modifiche, la definizione della classe privata per gestire l'evento e l'aggiunta del caso del nodo di tipo divisione nel metodo **addNode**, di cui non è riportato il codice. A questo punto l'aggiunta dell'operatore è terminata, e si possono fare i primi test per verificare che l'operatore sia presente e si comporti in modo corretto. In Figura 6.2 è mostrato un semplice esempio, sulla base di dati del Capitolo 4. Nella Figura 6.3 invece si può osservare un piano equivalente ottenuto applicando la definizione di ÷ come operatore derivato. Tale piano trova i valori di **FkNumFattura** delle fatture che hanno ordinato tutti gli articoli.

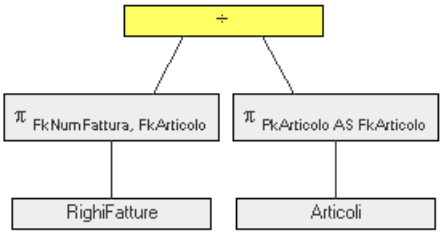

Figura 6.2 Piano logico con il nuovo operatore divisione

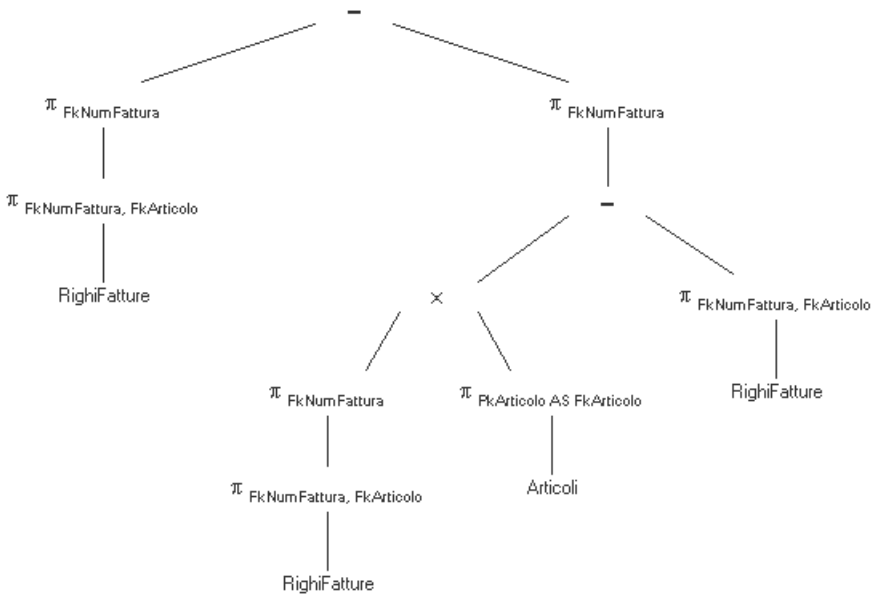

Figura 6.3 Piano logico con per eseguire la divisione senza l'operatore

6.2.3 La modifica per la generazione dell'SQL

Resta da definire la traduzione in **SQL** dell'operatore appena aggiunto. Nel Capitolo 5 è stato descritto in modo dettagliato l'algoritmo che genera l'interrogazione **SQL**. Per questo caso semplificato, esiste una traduzione che date due relazioni $R(A, B)$ e $S(B)$, calcola $R \div S$:

```
SELECT DISTINCT A
FROM            R x
WHERE NOT EXISTS
                (SELECT   *
                FROM S y
                WHERE NOT EXISTS
                        (SELECT    *
                        FROM       R z
                        WHERE      z.A = x.A
                          AND      z.B = y.B));
```

Si noti come nell'interrogazione appena mostrata, si usino delle clausole **WHERE NOT EXISTS** per rendere la quantificazione universale della definizione dell'operatore. Infatti:

$$R \div S = \{w \mid \forall s \in S.w \circ s \in R\} = \{w \mid \nexists s \in S.w \circ s \notin R\} =$$
$$\{w \mid \nexists s \in S.\nexists w \circ s \in R\}$$

Esistono altre possibili traduzioni, e una soluzione più efficiente è stata proposta in [5]. Si noti però come tale soluzione restituisca l'insieme vuoto nel caso in cui si divida per l'insieme vuoto, mentre secondo la definizione dell'operatore, in quel caso il risultato è l'insieme di tutti i valori di A. La soluzione proposta è la seguente:

```
SELECT      A
FROM        R
WHERE       B   IN    (SELECT B
                       FROM S)
GROUP BY    A
HAVING      COUNT(*) =
                      (SELECT    COUNT (*)
                      FROM       S);
```

Dato lo scopo didattico del sistema, si preferisce la prima traduzione, meno efficiente ma sempre corretta e coerente con la definizione dell'operatore. Tale traduzione presuppone che la divisione sia effettuata su due relazioni R ed S e quindi su due nodi foglia. Se non sono nodi foglia, si può trasformare il sottoalbero in una vista, secondo il meccanismo visto nel Capitolo 5. Si aggiungono quindi, nel metodo **generateViews**, le nuove condizioni per la creazione di una vista se il padre è un nodo di tipo ÷, e per il nodo stesso, si genera la traduzione appena mostrata, e nel caso avesse un padre, si trasforma in una vista anch'esso. Per la generazione delle viste dei sottoalberi è sufficiente aggiungere queste poche righe al metodo **generateViews**:

Listing 6.4 Codice Java per generare le viste degli operandi del nodo divisione

```
1 else if(thisNode.getTreeParent() != null &&
2   (thisNode.getLeft() != null || thisNode.getRight() != null)) {
3   if(thisNode.getTreeParent() instanceof LogicNodeDivision) {
4     view+=subTreeToSQL(thisNode);
5   }
6 }
```

Un breve commento:

- nella riga 1 si controlla che il nodo abbia un padre, e cioè che non sia il nodo radice.
- nella riga 2 si verifica che il nodo non sia una foglia, controllando che abbia almeno un figlio.
- nella riga 3 ci si chiede se il padre è un nodo del tipo **LogicNodeDivision**

– nella riga 4, stabilito che il nodo corrente non è il nodo radice, non è una foglia ed ha un nodo ÷ come padre, si genera la vista per il nodo corrente, operando della divisione.

Ora, stabilito che gli operandi sono foglie, essendo stati trasformati in viste, si aggiunge la traduzione in **SQL** del nodo stesso, sempre all'interno del metodo **generateViews**:

Listing 6.5 Codice Java per generare la traduzione SQL del nodo divisione

```
1  else if (thisNode instanceof LogicNodeDivision) {
2
3  String leftQuery = subTreeToSQL(thisNode.getLeft());
4
5  leftQuery = Utility.myReplaceAll("SELECT", "", leftQuery);
6  leftQuery = Utility.myReplaceAll("DISTINCT", "", leftQuery);
7  leftQuery = Utility.myReplaceAll("FROM", "", leftQuery);
8  String attributoNonInterno = leftQuery.substring(0,
9            leftQuery.indexOf(',')).trim();
10 String attributoInterno = leftQuery.substring(
11           leftQuery.indexOf(',') + 1, leftQuery.indexOf("\n"))
12          .trim();
13 String operandoEsterno = leftQuery.substring(
14           leftQuery.indexOf("\n") + 1, leftQuery.length()).trim();
15
16 String rightQuery = subTreeToSQL(thisNode.getRight());
17
18 rightQuery = Utility.myReplaceAll("SELECT", "", rightQuery);
19 rightQuery = Utility.myReplaceAll("DISTINCT", "", rightQuery);
20 rightQuery = Utility.myReplaceAll("FROM", "", rightQuery);
21
22 String attributo = rightQuery.substring(0,
23           rightQuery.indexOf("\n")).trim();
24 String operandoInterno = rightQuery.substring(
25           rightQuery.indexOf("\n") + 1, rightQuery.length())
26           .trim();
27
28 String corrOS = "OS" + tree.getNextSeqNumber();
29 String corrOI = "OI" + tree.getNextSeqNumber();
30 String corrOS2 = "OS" + tree.getNextSeqNumber();
```

```
31
32 view += Utility.alignTo(Constants.queryAlign,
33              "SELECT␣DISTINCT␣")
34              + attributoNonInterno + "\n";
35 view += Utility.alignTo(Constants.queryAlign, "FROM␣")
36              + operandoEsterno + "␣" + corrOS + "\n";
37 view += Utility.alignTo(Constants.queryAlign, "WHERE␣")
38             + "NOT␣EXISTS␣\n";
39 view += Utility.alignTo(Constants.queryAlign, "")
40             + Utility.alignTo(Constants.queryAlign, "(SELECT␣")
41             + "*␣\n";
42 view += Utility.alignTo(Constants.queryAlign, "")
43             + Utility.alignTo(Constants.queryAlign, "FROM␣")
44             + operandoInterno + "␣" + corrOI + "\n";
45 view += Utility.alignTo(Constants.queryAlign, "")
46             + Utility.alignTo(Constants.queryAlign, "WHERE␣")
47             + "NOT␣EXISTS␣\n";
48 view += Utility.alignTo(Constants.queryAlign, "")
49              + Utility.alignTo(Constants.queryAlign, "")
50              + Utility.alignTo(Constants.queryAlign, "(SELECT␣")
51              + "*␣\n";
52 view += Utility.alignTo(Constants.queryAlign, "")
53             + Utility.alignTo(Constants.queryAlign, "")
54             + Utility.alignTo(Constants.queryAlign, "FROM␣")
55             + operandoEsterno + "␣" + corrOS2 + "\n";
56 view += Utility.alignTo(Constants.queryAlign, "")
57              + Utility.alignTo(Constants.queryAlign, "")
58              + Utility.alignTo(Constants.queryAlign, "WHERE␣")
59              + corrOS2 + "." + attributoNonInterno + "␣=␣" + corrOS
60              + "." + attributoNonInterno + "\n";
61 view += Utility.alignTo(Constants.queryAlign, "")
62              + Utility.alignTo(Constants.queryAlign, "")
63              + Utility.alignTo(Constants.queryAlign, "␣␣AND␣")
64              + corrOS2 + "." + attributoInterno + "␣=␣" + corrOI
65              + "." + attributo + "))\n";
66 }
```

Segue un commento al codice appena mostrato:

– nella riga 1 si verifica che il nodo corrente sia del tipo **LogicNodeDivision**

– nella riga 3 si genera l'interrogazione per il sottoalbero sinistro, al fine di ricavarne i nomi degli attributi e dell'operando nella forma corretta.
– nelle righe 5-7 vengono eliminate le parole chiave dall'interrogazione appena generata.
– nelle righe 8-14 si estraggono i nomi dell'attributo anche nell'operando interno, dell'attributo non nell'operando interno e il nome dell'operando esterno. Per quanto fatto in precedenza, i nomi degli operandi sono viste, precedentemente generate.
– nelle righe 16-26 si genera l'interrogazione dell'operando esterno e in modo analogo a quanto fatto per l'operando interno, si estraggono il nome dell'attributo e dell'operando.
– nelle righe 28-30 vengono stabiliti i nomi delle variabili di correlazione da utilizzare nell'interrogazione.
– nelle righe 32-66 si compone l'interrogazione, avendo a disposizione tutti i dati.

A questo punto la traduzione è completa, ed è possibile eseguire piani logici che utilizzino l'operatore definito in questo Capitolo.

6.2.4 Esempio di utilizzo dell'operatore di divisione

Tornando all'esempio di Figura 6.2, con la traduzione in **SQL** definita, possiamo far generare l'interrogazione ottenendo:

```
CREATE VIEW LTV0 AS
SELECT DISTINCT FkNumFattura, FkArticolo
FROM            RighiFatture
;
CREATE VIEW LTV1 AS
SELECT DISTINCT PkArticolo AS FkArticolo
FROM            Articoli
;
SELECT DISTINCT FkNumFattura
FROM            LTV0 OS0
WHERE NOT EXISTS
                (SELECT   *
                FROM LTV1 OI1
                WHERE NOT EXISTS
                        (SELECT    *
                        FROM       LTV0 OS2
                        WHERE      OS2.FkNumFattura = OS0.FkNumFattura
                          AND      OS2.FkArticolo = OI1.FkArticolo))
;
```

Eseguendo il piano si ottiene il risultato corretto. Tale piano trova le fatture in cui sono stati ordinati tutti i prodotti.

6.3 Conclusioni

Si è mostrata la modificabilità del sistema indicando esattamente quali metodi sono responsabili delle funzionalità che probabilmente si vorranno aggiornare spesso, e si è fornito un esempio di estendibilità, mostrando nel dettaglio come si aggiunge un nuovo operatore logico. Tale esempio vuole essere una guida per future estensioni del sistema. Per quanto riguarda l'operatore realizzato, i possibili miglioramenti sono due:

– ridurre l'uso delle viste, in modo analogo a quanto è possibile osservare per il caso dell'operatore γ. In generale, quando nella visita BFS rovesciata si arriva al nodo $\div$, i sottoalberi sono sempre traducibili senza introdurre ulteriori viste, e cioè il metodo **subTreeToSQL** genera correttamente l'inter-

rogazione per i sottoalberi. Si potrebbero quindi annidare le interrogazioni ottenute per poi riscrivere l'interrogazione stessa per eliminare le *sottoselect*, ottenendo una traduzione più compatta.

– generalizzare l'operatore perchè abbia la stessa semantica dell'operatore algebrico definito prima.

Capitolo 7

CONCLUSIONI

È stato realizzato un editore grafico di piani logici e fisici, scritto in Java come estensione del DBMS relazionale JRS, a scopo didattico. Permette per esempio agli studenti dei corsi di basi di dati di formulare interrogazioni in algebra relazionale o con operatori fisici. L'interfaccia grafica è stata realizzata in modo da guidare l'utente nella formulazione dell'interrogazione, e mostra per ogni operatore chiaramente il tipo del risultato, consultabile anche visualizzando le informazioni di un singolo nodo, e quali parametri vanno dati e in che ordine, attraverso appositi *menu contestuali*. Importanti sono anche le etichette dei nodi, che mostrano i parametri nella stessa forma in cui appaiono nei libri di testo. Le interrogazioni così formulate, visualizzate graficamente come alberi, possono poi essere eseguite. L'esecuzione delle interrogazioni avviene in modo diverso per i piani logici e fisici:

– nei piani logici, si converte il piano in un'interrogazione **SQL**, da far eseguire poi al JRS. In presenza di piani complessi, si divide il piano in più parti ricorrendo alla creazione di *viste* per la traduzione di sottoalberi. L'utente può a scelta visualizzare l'interrogazione generata o eseguirla direttamente, senza vederla, eseguendo di fatto il piano che ha disegnato. È inoltre possibile eseguire sottoalberi, in modo da poter suddividere l'analisi di un piano logico anche complesso come analisi di più sottoalberi, senza doverlo trasformare.

– nei piani fisici, si genera un albero di iteratori, utilizzando gli operatori fisici presenti nel sistema. Una volta generato tale albero, con gli operatori correttamente annidati e con i parametri corretti, è sufficiente iterare sull'operatore radice dell'albero per ottenere il risultato dell'interrogazione. Si noti che allo scopo di rendere possible quest'operazione, diversi operatori fisici del JRS sono stati rivisti, perchè a volte non funzionavano correttamente in presenza di piani dalla struttura diversa rispetto a quelli che genera l'ottimizzatore.

Lo scopo del progetto non si limita a fornire agli studenti uno strumento per formulare e sperimentare con piani logici e fisici, ma è anche uno strumento di supporto ai docenti di basi di dati, permettendo di mostrare piani, evidenziare dei nodi e spostare dei sottoalberi, oltre che compiere le operazioni descritte in precedenza. Inoltre non vi sono sistemi commerciali o didattici con funzionalità simili, il che da un lato ha reso più difficile la realizzazione non avendo esempi da usare come punti di riferimento, ma dall'altro rende il risultato del lavoro più interessante. Si è mostrato come il sistema sia modificabile ed estendibile, fornendo anche un esempio completo di estensione. Altre possibili estensioni del sistema sono:

– aggiungere la stima dei costi per i nodi dei piani fisici, utilizzando le stesse formule del JRS. Tale aggiunta permetterebbe di vedere le stime dei costi per i piani creati, dando indicazioni sulla loro efficienza. Il JRS permette di visualizzare graficamente i piani di accesso generati attraverso l'ottimizzatore, pertanto il confronto tra i piani generati e quelli creati con l'editore è immediato.
– convertire un albero di iteratori creato dall'ottimizzatore in un piano fisico dell'editore. Tale aggiunta permetterebbe di modificare facilmente i piani generati dall'ottimizzatore per un'interrogazione, oltre che vederli in maggior dettaglio.
– ammettere sottoalberi nei piani, caricati da file e visualizzati come un unico nodo, che all'occorenza può essere espanso mostrando il sottoalbero.

– in aggiunta al punto precedente, si può associare il nodo che rappresenta un sottopiano al corpo di una funzione, con la possibilità di dare parametri al nodo. Tale funzionalità sarebbe simile al **Create Function** di **SQL**.

Il JRS così esteso diviene un'utile strumento di supporto alla didattica per facilitare l'apprendimento di concetti importanti nello studio delle basi di dati.

BIBLIOGRAFIA

[1] Albano A. Basi di dati di supporto alle decisioni. Appunti delle lezioni del corso di Sistemi informatici direzionali, Università di Pisa, 2010.

[2] Albano A., Ghelli G., Orsini R., *Fondamenti di basi di dati*, Zanichelli, 2005.

[3] Codd E. F., *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, June 1970.

[4] Codd E.F. , Serious Flaws in SQL, in: *The Relational Model for Database Management*. Version 2, Addison-Wesley , 1990, pp. 371-389.

[5] Matos V. M., A Simpler (and Better) SQL Approach to Relational Division, *Journal of Information Systems Education*, Volume 13, Issue 2, 2002, pp. 85-87.